2002

High-level clock construction for low power.

Changjun. Kang
University of Windsor

HIGH-LEVEL CLOCK CONSTRUCTION FOR LOW POWER

by

Changjun Kang

A Thesis
Submitted to the Faculty of Graduate Studies and Research
Through the Department of Electrical and Computer Engineering
In Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science
At the University of Windsor

Windsor, Ontario, Canada

2002

ABSTRACT

Clock network is the heaviest load in synchronous VLSI systems. It accounts for large portion of the total power dissipation. Our work concentrates on high-level optimization of the power of clock network, which is a relatively new area. Our work includes two parts: activity-sensitive clock design for low power and low power clock based on clock frequency reduction.

In the activity-sensitive clock design, we introduce the term of *node difference* based on module activity information, and show its relationship with power consumption. *Merging power* is used to measure the power cost of merging two nodes. A binary clock tree is built based on the *merging power* between different modules to optimize the power consumption due to interconnections (i.e., clock gating signals and clock edges). We also develop a method to determine the gating signals with least transitions. After the clock tree is constructed, we apply a local optimization on gating signals to further reduce the power consumption.

In the clock frequency reduction, we propose a high-level power optimization scheme with two techniques: operator chaining, multiple clocks. Chaining the operators with shorter delay allows the use of a lower clock frequency. With multiple clocks, the operators with longer delay can be driven by another clock with lower frequency. These techniques are combined with clock gating to reduce clock power consumption. Our experiments with benchmarks show that the clock power reduction rate and total power savings are around 46% and 15%, on average, respectively, with a little or no performance degradation.

These methods can be used in behavioral or RT (Register Transfer) level synthesis, combining with other low power techniques, to generate low power circuits. Appendices are attached in the thesis including the VHDL (VHSIC Hardware Description Language) codes and CDFG (Control Data Flow Graph) files of the benchmarks, and part of the C&C++ source codes of the algorithms.

ACKNOWLEDGEMENT

I would like to extend my appreciation to my thesis supervisor Dr. Chunhong Chen, who has directed me throughout my whole research work. His insight in the related academic field, and many fruitful discussions were very helpful in the process of my research work. I would also like to thank Dr. M. Ahmadi, Dr. A. Jaekel, and Dr. S. Erfani for their constructive comments.

Finally, I would like to thank my family especially my wife for their support and understanding.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure **Page**

CHAPTER 1
INTRODUCTION

1.1 Necessity of low power

VLSI (Very Large Scale Integrated-circuit) has become the heart of computers, office machines, and all kinds of appliances for many years. As the technology progresses, the complicated IC (Integrated Circuit) may integrate millions, or even hundreds of millions of transistors. The power dissipated by the small silicon piece could be incredible. The low power design is developed to reduce the power consumption of ICs. The main driving forces behind it are: portable system, thermal requirement, reliability and environmental concerns.

1.1.1 The popularity of portable devices

The major concerns of VLSI designers were area, performance, reliability, cost etc in the past. Low power was only of secondary importance. In recent years, the situation has begun to change. Power consumption has received more and more attention. One of the primary driving force behind it is widespread use of portable electronics. The portable electronics ranges from portable laptop, camcorder, PDA (Personal Digital Assistant) to cellular phone etc. These electronic devices are powered by battery and hence battery life is a very critical parameter in the evaluation.

The advances in battery technology have not been able to keep up with the growth in energy consumption requirement of portable electronics. The most popular batteries used in these electronic devices are Nickel-Cadmium (Ni-Cd), Nickel-Metal-Hydride (Ni-MH) and Lithium Ion batteries. Today's typical Ni-Cd battery has energy density of about 50 Watt-hours/kilogram [1]. Ni-MH and Lithium Ion batteries have energy density of around 60, 115 Watt-hours/kilogram respectively. If we want to provide 10 hours of operation to 40W device, the device has to be equipped with 8KG Ni-Cd, or 6.7KG Ni-MH, or 3.5KG Lithium Ion battery. The weight and cost of battery pack prevent the popularity of these portable devices without adoption of low power design techniques. In most

portable devices, the integrated circuits account for large portion of total power dissipation. Thus, a lot of efforts have been made toward developing low power VLSI design techniques.

1.1.2 Thermal requirement

The power consumed by integrated circuits is dissipated mostly in the form of heat. In order to keep the chip under acceptable temperature, some cooling techniques are needed. High temperature results in degradation or malfunction of chip. Contemporary microprocessor dissipates as much as 15-30W at 100-200MHz clock rate [2]. It is estimated that a $10cm^2$ microprocessor clocked at 500MHz consumes about 300W [3]. The power dissipation in microprocessor increases roughly linearly in proportion to its die size and clock frequency. These processors with large power consumption demand expensive package and cooling system. Studies showed that every 10°C increase in operating temperature almost doubles the failure rate of the component [4]. Unless power consumption is significantly reduced, the potentially powerful microprocessors are unlikely to work properly.

1.1.3 Reliability

The peak and average power consumption affect the operation of integrated circuits. Heat accelerates *electromigration* that is a mass transportation of metal atoms, leading to electrical cuts or shorts of metal wire lines. Hot carrier effects in MOS transistors (charge trapping in the oxide and interface trap at the Si/SiO_2 interface resulting in variation of threshold and slower electron mobility in the channel) are related to switching rates of transistors [5]. Reducing the peak and average power dissipation of IC normally improve the reliability.

1.1.4 Environmental concerns

Small power consumption has less impact on global environment. According to an estimate by the U.S Environmental Protection Agency (EPA), 80% of the total office equipment electricity consumption comes from computing equipment, a large part of which is due to the equipment consuming current even when unused [6]. EPA's *Energy Star* program is therefore launched. *Energy*

Star specifies the standards for power-efficient PCs. The power management techniques have been widely applied on desktops and laptops since then.

1.2 Sources of power dissipation

There are 3 sources of power dissipation in CMOS circuits:

1, Capacitive switching power

2, Short-circuit power

3, Static power

1.2.1 Capacitive switching power

Capacitive switching power is consumed on charging and discharging the parasitic capacitance in circuit. The capacitances in circuits include transistor's capacitances and net distributive capacitance. As Figure 1.1 shows, MOS transistor has gate-source capacitance Cgs, gate-drain capacitance Cgd, gate-substrate capacitance Cgb, source-substrate capacitance Cbs, and drain-substrate capacitance Cbd.

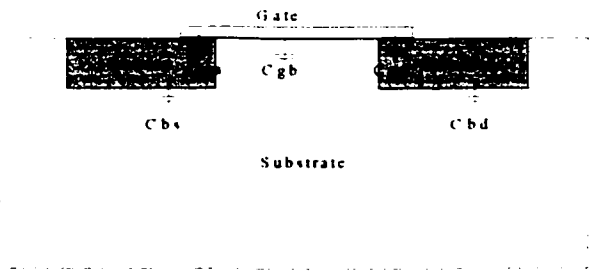

Figure 1.1 Capacitances in CMOS transistor

A CMOS inverter shown in Figure 1.2 demonstrates capacitive switching power. The C_L is the aggregate capacitor including capacitors associated with NMOS and PMOS transistor and net distributive capacitance.

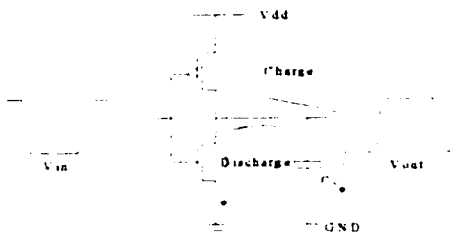

Figure 1.2 Charging and discharging CMOS inverter

In the initial state in which V_{in} is logically 1 and V_{out} is logically 0, C_L is fully discharged. When the input undergoes a falling transition, PMOS transistor switches on and NMOS transistor switches off. The power supply charges C_L until V_{out} becomes V_{dd}. During the charging process, the total energy is $C_L V_{dd}^2$, half of which is stored in C_L and the other half is consumed on PMOS and interconnect. When the input goes from low to high, PMOS transistor turns off and NMOS transistor turns on. This is a discharging process in which V_{out} finally becomes 0. The $0.5C_L V_{dd}^2$ energy that was stored in C_L is dissipated on NMOS transistor and interconnects.

Assuming that the inverter is part of a synchronous circuit running at clock frequency f, and that rising transition is half of the total transition. The above analysis suggests the formulation of capacitive switching power [26].

$$p_c = 0.5C_L V_{dd}^2 Nf \qquad (1.1)$$

where N is the average of expected number of transitions (rising and falling) per clock cycle. We refer to N as *switching activity*. The product of *switching activity* and load capacitance C_L is termed *switched capacitance*. In static CMOS technology, the capacitive switching power is the dominant part of the total power. As a result, most power estimation and optimization techniques concentrate on how to reduce the capacitive switching power.

1.2.2 Short-circuit power

Short-circuit power is due to the current flowing from V_{dd} to ground in transition time. The inverter current during transition is shown Figure 1.3. When the input of an inverter rises from 0 to V_{dd} or vice verse, there is a period of time in which both NMOS and PMOS transistor are on. The current drawn from supply to ground is referred to as *short-circuit current*. The short-circuit power consumption is proportional to the input ramp time. The maximum *short-circuit current* appears when no load exists. This current decreases with the load. Different formulations of *short-circuit current* were derived [7, 8]. Short-circuit power is roughly proportional to rise and fall delay. For example, the short-circuit power dissipation of a CMOS inverter can be approximated by [9]

$$P_s = K(V_{dd} - 2V_T)^3 \tau N f \qquad (1.2)$$

where K is a constant that is associated with transistor size and the technology, V_T is the average magnitude of the threshold voltage of the NMOS and PMOS transistors, τ is the rise/fall time, N is the number of transitions of the inverter and f is the clock frequency.

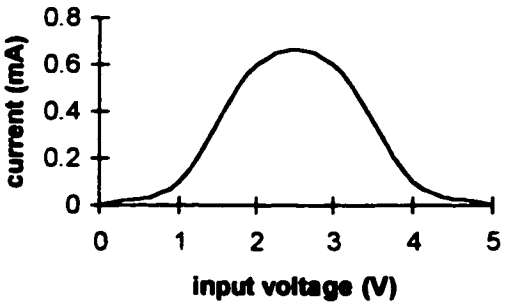

Figure 1.3 Current-voltage diagram of inverter

The short-circuit power dissipation can be controlled by reducing the rise/fall time to all gates. The short-circuit power dissipation for well-designed circuits is normally less than 15% of total *dynamic power dissipation*. The term *dynamic power dissipation* refers to the sum of short-circuit power dissipation and capacitive switching power dissipation.

1.2.3 Static power

Leakage power is part of static power. Leakage power dissipation is caused by the current of reverse-biased diodes and sub-threshold current. So the leakage power dissipation can be written by

$$P_L = (I_{diode} + I_{subth})V_{dd} \qquad (1.3)$$

In the above equation, I_{diode} is the anti-biased current of diodes. The diodes exist between the source/drain and the substrate. I_{subth} refers to the currents arising due to the fact the transistors that are off conduct some non-zero current. Leakage power dissipation is very important for devices that are in an idle state in most of time.

In CMOS circuits, static power consumption can also result due to degenerated voltages level at the inputs to a static gate, or due to selector, or bus-conflicts where multiple drivers attempt to drive a signal to different logic values. Such situations are undesirable, and are typically avoided through proper circuit design techniques.

1.3 Low power techniques overview

Most of low power design techniques focus on reducing the capacitive switching power. According to equation 1.1, power can be optimized by reducing supply voltage, switched capacitance or clock frequency. It is necessary to make a distinction between power dissipation and energy dissipation that are often used interchangeably [10]. In the system powered by battery, the duration of working time is limited by the amount of energy stored in the battery. Power is the rate at which energy is consumed. Power dissipation is a critical parameter for cooling and packaging design, whereas energy is a measurement of battery life. If a technique makes a computation performed at half power of the previous technique, with twice time to complete it, the energy consumed by the two techniques gives no difference.

1.3.1 Voltage

Because supply voltage has quadratic effect on power dissipation, voltage reduction is the most effective way to reduce the power. Unfortunately, voltage reduction incurs increase of delay [10].

$$delay = k\frac{V_{dd}}{(V_{dd} - V_T)^2} \qquad (1.4)$$

where k is a constant. V_T is the threshold voltage. As indicated by equation 1.4, the delay increases drastically when V_{dd} approaches the threshold voltage V_T. Normally V_{dd} is set at least two or three times V_T. Another penalty of voltage reduction is reducing the circuit's noise margin, which makes the circuit susceptible to noise-related failure [11].

One approach of reducing voltage without throughput penalty is to change the threshold V_T. Reducing V_T allows supply voltage to be scaled down without loss in speed. The minimum V_T is controlled by the specified noise margin and subthreshold current. Another approach is to adopt faster logic and architectural blocks such as *Wallace* multiplier instead of *Array* multiplier. Although the *switched capacitance* increases, it enables reducing supply voltage to decrease power dissipation [12, 13, 14]. Multiple supply voltages can be applied to optimize the power dissipation ,where the lower supply voltage is used on non-critical path, resulting in little or no penalty of speed [15, 16].

1.3.2 Switched capacitance

Switched capacitance is product of *switch activity* and load capacitance in equation 1.1. Switched capacitance can be reduced by different techniques from physical design level to system level.

At physical design level, the techniques involved in floor planning, routing and clock generation reduce the load capacitance; At circuit level, reordering gate inputs and transistor sizing

can be used to optimize the power dissipation [17]. At logic level, Don't-care optimization, path balancing, factorization and technology mapping can reduce the power dissipation of combinational logic circuit. Encoding, retiming, gated clock and precomputation are used in optimization of the power consumption of sequential logic circuit; At the architectural level, power management, architectural transformation, multiple clocks have been used to reduce switched capacitance. At system level, the instructions of CPU have an impact on power dissipation. An appropriate choice of instructions in the generated code can lead to a reduction in the power cost [18].

1.3.3 Clock frequency

Reducing clock frequency leads to power reductions while the energy consumption for a specific computation is not changed. It also results in performance degradation that is undesired. That is why reducing clock frequency is not a useful means in low power design, most of which require energy saving at the same moment. However, when the circuit stays idle for most of the time with only very little computation to be performed, dynamically reducing the operating frequency helps eliminate unnecessary power and energy consumption. Little penalty on performance is incurred. This technique is commonly used as power management in microprocessors [19].

1.4 High-level techniques for low power

High-level synthesis (behavioral synthesis or architectural synthesis) refers to the process of transforming a functional or behavioral specification of a design into a structural RTL (Register Transfer Level) implementation. High-level synthesis often involves some subtasks such as scheduling, binding, resource sharing and clock selection etc. Those high-level techniques have larger effect than the low-level techniques on power consumption. It's necessary to introduce these techniques since some concepts are closely associated with our work.

Before introducing these high-level techniques, CDFG (control data flow graph) needs to be explained.

Level	System	RT	Logic	Circuit	Physical
Power Saving	50-90	30-60	20-40	10-30	5-15

Table 1.1 The effect of different level techniques for low power

1.4.1 Control data flow graph (CDFG)

During high-level synthesis, some metrics such as area, performance, power and testability are either constraints or co-objectives. The hardware description languages like VHDL, Verilog are often used in behavioral description. These hardware description languages not only support logic, RT level description of circuit, but also support some behavioral description. The behavioral description includes some primitive data type (integers, floating point numbers, enumerated types, etc.), control and loop, and sub-routines. In general, CDFG can be categorized into two application domains. One is data-flow intensive domain; the other is control-flow intensive domain. DSP is normally data-flow intensive since most of nodes are arithmetic computations. The protocol and controller are generally control-flow intensive. Table 1.2 gives an example.

```
architecture diffeq of diffeq is
begin
P1 : process (Aport, DXport, Xinport, Yinport, Uinport)
   variable x_var,y_var,u_var, a_var, dx_var: integer ;
   variable x1, y1, t1,t2,t3,t4,t5,t6: integer ;
begin
  x_var := Xinport;  a_var := Aport; dx_var := DXport; y_var := Yinport; u_var := Uinport;
--    Xinport <= x_var + a_var;
   while (x_var < a_var) loop
      t1 := u_var * dx_var;
      t2 := 3 * x_var;
      t3 := 3 * y_var;
      t4 := t1 * t2;
      t5 := dx_var * t3;
      t6 := u_var - t4;
      y1 := u_var * dx_var;
      u_var := t6 - t5;
      y_var := y_var + y1;
      x_var := x_var + dx_var;
   end loop;
  Xoutport <= x_var;
  Youtport <= y_var;
        t1 := x_var + y_var;
  Uoutport <= t1;
end process P1;
end diffeq;
```

Table 1.2 VHDL description of differential equation

The VHDL code is compiled by some tools. The output of the compilation is CDFG. Figure 1.4 shows the CDFG of differential equation. In this thesis, CDFG file is obtained by two steps. First, we use *VDT2.7* to convert VHDL code into a file suffixed *IF*, which represents *Intermediate Format* [20]. Then the corresponding CDFG file is generated by *CDFGGEN* [21].

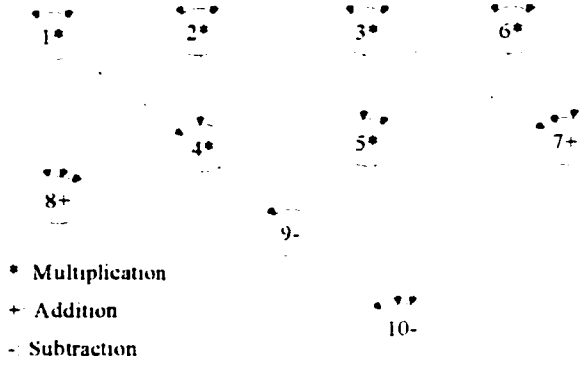

Figure 1.4 CDFG of differential equation

In CDFG, operations are represented by nodes, while data, control signal are represented by edges. For each node, it has preceding edges and successive edges. For each edge, it has starting node and ending node. Based on the dependence between the nodes, the sequence of execution could therefore be obtained.

1.4.2 Scheduling, binding and resource allocation

Scheduling refers to the process of assigning operations in the behavioral description to *control steps* (clock cycle). Scheduling has great impact on the performance, area, and power of the final system. In practice, some constraints such as total delay, resource can be applied on the scheduling. Time-Constrained Scheduling (TCS) is the scheduling in which the total delay specified by designer must be satisfied. In Resource-Constrained Scheduling (RCS), the resource (functional units, registers) available to implement the design is fixed. The objective is to minimize the number of control steps [22].

As-Soon-As-Possible (ASAP) and As-Late-As-Possible (ALAP) scheduling are the two simplest scheduling algorithms used in high-level synthesis. ASAP scheduling schedules each operation at the earliest possible control step. ALAP scheduling, on the contrary, schedules each operation at the latest possible control step. *List scheduling* is a constructive heuristic algorithm solving RCS problem [23]. *Force-Directed Scheduling* is a popular algorithm that solves TCS problem [24]. Scheduling affects the peak power and power distribution of the system [13].

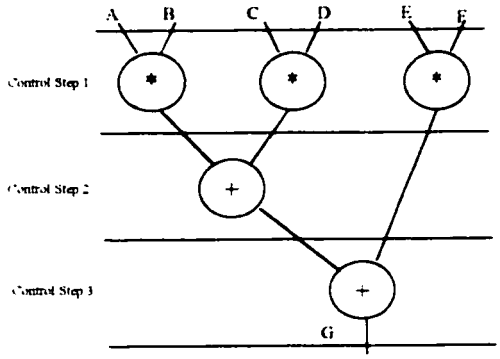

Figure 1.5 Schedule for the expression "G=AB+CD+EF"

Binding refers to the process of assigning operations to functional units, assigning values to storage units, and interconnecting those components to form a complete data path. In binding, only the functional unit template, and not a specific instance, is associated with each operation. For example, a multiplier can be either implemented by Array multiplier or Wallace multiplier. Each distinct implementation for the same operation may have different area, delay, and power characteristics. Wallace multiplier is faster than Array multiplier while its switched capacitance and power dissipation are also larger. The selection of implementation for each operation in CDFG is crucial for power saving. The operations on non-critical path of CDFG are bound to slower functional unit, which also tend to be more switched capacitance efficient. The operations on critical path are still mapped to fast functional units [12, 13, 14]. Figure 1.6 gives an example. In Figure 1.6a, node 3 is mapped to Wallace multiplier while it is mapped to Array multiplier in Figure 1.6b. Since node 3 is not on critical path. The performances (delays) under both bindings are identical. The binding of Figure 1.6b leads to less power dissipation than Figure 1.6a.

The slack on non-critical path also enables the technique of multiple supply voltages [25]. The operations on non-critical path are supplied with lower voltage. High-low and low-high level converters are the penalty for the technique. The choice between binding and multiple supply voltages is made depending on the constraints.

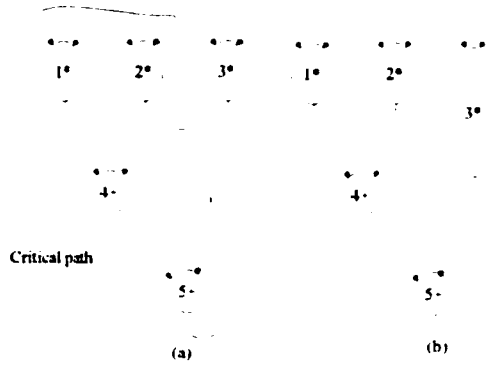

Figure 1.6 Minimizing power consumption by binding

Resource is here referred to as the collection of functional instances. For example, a resource has 3 Ripple-Carry adders and 2 Array multipliers. Resource allocation maps the operations and variables in the CDFG to the specific functional instances and registers, and defines interconnection among the functional instances and registers, to form the RTL implementation [26]. The temporal correlation among inputs for functional units or variables for registers can be exploited to reduce *switch activity* that is average number of transitions per clock cycle [27, 28, 29]. Hence, switched capacitance is decreased. It's also possible to exploit the regularity on data bus to minimize the interconnection power [30].

1.5 Summary of contributions

Two high level methods are proposed on low power clock design. They are activity-sensitive clock design for low power and low power clock based on clock frequency reduction respectively.

In the activity-sensitive clock design, *node difference* is proposed to measure the similarity between two activity patterns. *Merging power* considers the effects of both clock edge and control signal. *Merging power* is used to evaluate the power cost of merging two nodes. A method is

developed to determine control signal with least transitions. In clock tree construction, all possible *merging powers* are computed and sorted in ascending order at each level. Two nodes with smallest *merging power* are paired to obtain their parent node at upper level until all nodes are merged. Clock tree is built up in bottom-up manner. After the clock tree is constructed, a technique call *local ungating* is applied to further reduce the clock power consumption in top-down manner.

In the clock frequency reduction, *operator chaining* and *multiple clocks* are proposed to reduce clock power. By chaining the primitive operators with short delays, the complex operator with long delay is created to realize the functions of its member operators. A global clock with lower frequency can be applied on circuits with very limited effect on the performance. Thus, clock power is reduced due to lower clock frequency. In *multiple clocks* scheme, the operators with long delays are driven by the low frequency clock while the operators with short delays are still clocked at high frequency. The clock power is saved by partially reducing clock frequency.

The above techniques are carried out on CDFG which is behavioral description of circuits. Therefore, the estimation of lower bound of clock power consumption can be made on the early stage of design.

CHAPTER 2
PROBLEMS IN CLOCK GENERATION

In a synchronous digital system, the clock network controls the timing of data transfer. It can dramatically affect the performance and reliability of the system. Many research works have been done on the topics such as zero-skew which means clock arrives at each module at the same time, minimal delay of clock, minimal wire length of clock, and clock power consumption. In this chapter, interconnect delay is firstly introduced followed by clock skew and zero-skew clock construction. At the end, some design techniques for low power clock are briefly explained.

2.1 Interconnect delay and its models

Clock network is basically interconnection inserted by some buffers on the chip. In CMOS design, the delay time of clock is dominated by the rise/fall time. Obviously, longer rise/fall time makes switching time longer in synchronous elements, and therefore, the clock cycle period must be longer in order to assure the time period for processing in combinational logic between the synchronous elements. In order to make clock rate faster , the delay time in a clock network must be minimized. The inserted buffers make the clock transition sharp so that the clock delay can be reduced [31].

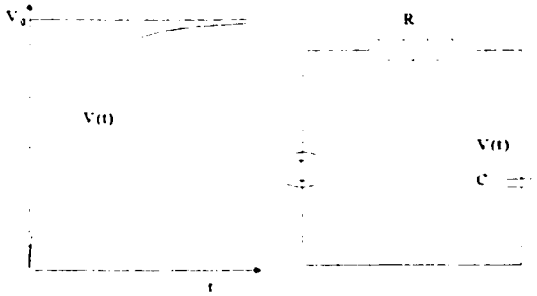

Figure 2.1 Response of simple RC circuit

Figure 2.1 illustrates the simple RC circuit and its response. The output response the simple RC circuit for step input is given by equation 2.1.

$$V(t) = V_0(1 - e^{-t/RC}) \tag{2.1}$$

$$T_d = \tau = RC \tag{2.2}$$

$$T_{50\%} \approx 0.7RC \tag{2.3}$$

Several delay definitions are used to evaluate the delay of circuits. Time constant τ is most often used as a measurement of the delay T_d. $T_{50\%}$ is the delay when $V(t)$ reaches half value of V_0.

There are two types of delay models available for interconnection. Lumped capacitance delay model treats the distributive capacitors and resistor as a simple RC circuit, which consists of one lumped capacitor and one resistor. The advantage of lumped delay model is that the delay calculation is quite easy. The result of lumped delay model is not accurate because the interconnection is inherently distributive RC line.

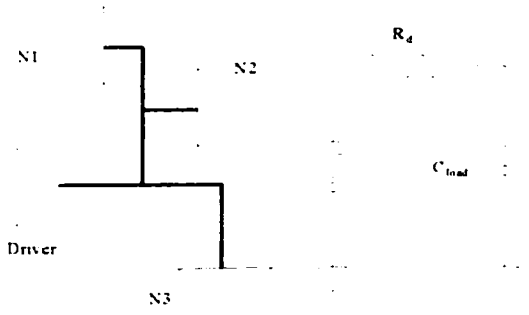

Figure 2.2 Lumped capacitance delay model

In Figure 2.2, R_d is driver's resistance. C_{load} is the sum of total interconnection capacitance C_{int} and buffers' input capacitance C_g. The delay T_d for step input can be expressed as follow:

$$T_d = R_d \cdot C_{load} = R_d \cdot (C_{int} + C_g) \tag{2.4}$$

The weaknesses of lumped capacitance delay are:

1) Good approximation only when driver resistance>>interconnect resistance.
2) All sinks have equal delay.

Distributed RC circuit can model interconnection more accurately than lumped capacitance model. The interconnect is divided into some segments, each of which can be modeled by L, T or Π circuit [32]. Figure 2.3 shows distributed RC circuit models.

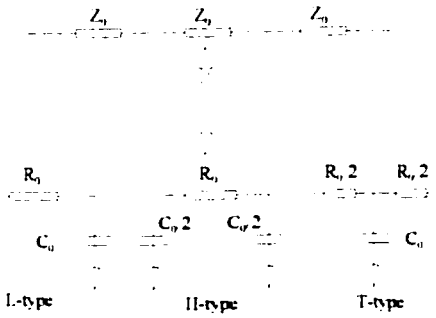

Figure 2.3 Distributed RC circuit models

In the distributed RC circuit, the calculation of delay at each node is a classic problem. Elmore delay is widely adopted to approximate the actual $T_{50\%}$ [33]. Elmore delay assumes that the circuit has a monotonic output response under unit step input. The basic idea of Elmore delay is to approximate median of $v'(t)$ by mean of $v'(t)$ that is rate of change of output $v(t)$. Elmore delay is defined as equation 2.5. The curves of $v(t)$ and $v'(t)$ are shown in Figure 2.4. It can be proved that $v'(t)$ reaches its median value over the monotonic response when t is $T_{50\%}$ by the definition of median.

$$T_{elm} = \int_0^{\infty} v'(t)t\,dt \tag{2.5}$$

$$\int_0^{T_{50\%}} v'(t)dt = \int_{T_{50\%}}^{+\infty} v'(t)dt = 0.5V_0 \tag{2.6}$$

where V_0 is the final value of $v(t)$. Elmore delay is not $T_{50\%}$ in general.

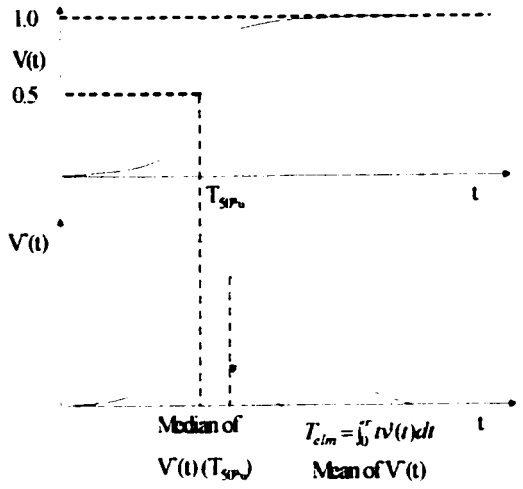

Figure 2.4 Elmore delay for monotonic response

The reason why Elmore delay is widely adopted on interconnections is that it is easier to compute analytically in RC trees. The Elmore delay for a RC tree is given by [34]

$$T_{elm(i)} = \sum_k R_{ki} C_k \qquad (2.7)$$

where $T_{elm(i)}$ is the Elmore delay at the node i. R_{ki} is the resistance of common path between path P_i and path P_k. Path P_i is the path from input to node i. C_k is the capacitance at node k. Figure 2.5 gives an example of how to calculate Elmore delay. As equation 2.7 indicates, Elmore delay can be computed recursively in linear time. This fact significantly reduces the time cost on computing delay. The disadvantages of Elmore delay are:

1) Low accuracy, especially poor for slope computation
2) Inherently cannot handle inductance effect.

For the interconnection, buffers not only drive the signal but also themselves are the load. The load caused by buffer is an active load, value of which is affected by biased voltage and current. For example, the gate-source capacitor C_{gs} of a MOSFET is related to gate and source voltage. Both

lumped capacitance model and distributed RC circuit model are passive network while interconnection is actually active network. The inaccuracy of calculating delay is unavoidable.

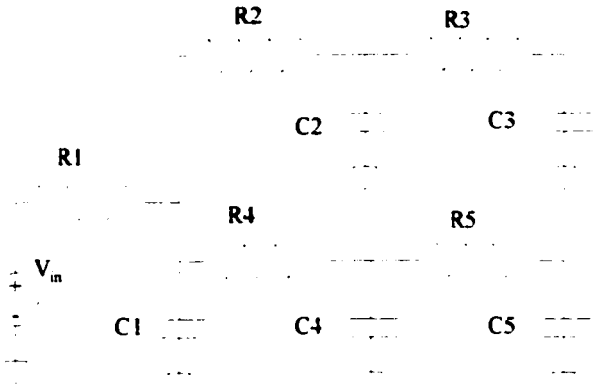

Figure 2.5 An example of calculation of Elmore delay

2.2 Zero-skew clock construction

In a synchronous digital systems, the global clock determines the time of data moving in and moving out of the registers. Most synchronous digital system consists of cascaded banks of sequential registers with combinational logic locating between a pair set of registers. The combinational logic circuits realize the functions such as multiplication, addition etc. Hence, the combinational logic is referred to as functional unit. The output of functional unit is data which sets up in the banks of registers and is ready to leave the registers until the triggering transition of the clock. Once the data leaves the registers, it propagates through next combinational circuit if available or becomes the output of the system. The synchronous digital system is basically composed of the following subsystems:

1) The memory storage elements
2) The logic elements
3) The clock circuitry and distribution

The clock network must be carefully distributed in order to make the synchronous system work properly. The clock also has impact on the performance of the system.

2.2.1 clock skew

A schematic of a generalized synchronized data path is present in Figure 2.6. The data path consists of two registers and a combinational logic.

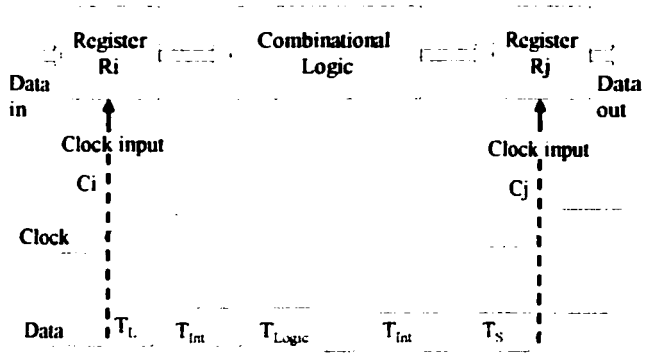

Figure 2.6 Timing diagram of clocked data path

In Figure 2.6, C_i and C_j are the clock inputs driving the front register and the back register, respectively, and both originate from the same clock signal source. The clock delay of the front clock signal T_{ci} and the back clock signal T_{cj} define the time reference when the data signals begin to leave their respective registers. The clock signals are used to synchronize each register. The difference in *delay between two sequentially adjacent clock paths*, as shown in Figure 2.6, is the clock skew T_{skew}. The temporal skew between the delays of different clock signals is only relevant to sequentially adjacent registers making up a single data path. Thus, system-wide clock skew between non-sequentially adjacent registers has no effect on the performance and reliability of the synchronous system. If the clock signals arrive registers at the same time, it is called *zero-skew* clock. Clock skew is expressed by

$$T_{skew} = T_{ci} - T_{cj} \tag{2.8}$$

The minimum allowable clock period (maximum clock frequency) is limited by the clock skew and the total propagation delay [35]. The reason is that the output of combinational logic has to be stable and latched in the back register before the next triggering clock transition (rise/fall).

$$T_{cp}(\min) > T_{pd} + T_{skew} \tag{2.9}$$

$$T_{pd} = T_L + T_{Logic} + T_{Int} + T_S \tag{2.10}$$

where T_{pd} is total propagation delay between the front register and the back register, T_L is the needed time for the data signal to leave the front register, T_{Logic} is the time to propagate the combinational logic circuit, T_{Int} is the time spent on interconnect, and T_S is the data set-up time of the back register. Clock skew can be positive or negative depending on whether C_j leads or lags C_i. Equation 2.9 shows clock skew affects the highest attainable clock frequency.

The maximum attainable clock frequency is determined by the combinational logic block with largest total propagation delay and its corresponding clock skew. In the case of negative clock skew, the clock skew must be less than the time required for the data leave the front register, propagate the combinational logic and interconnect, and set up at the back register.

$$|T_{skew}| \le T_{pd} \qquad \text{for} \quad T_{ci} < T_{cj} \tag{2.11}$$

Negative clock skew can be used to improve synchronous performance [36]. By forcing C_i to lead C_j at each critical data path, excess time is shifted from the neighboring less critical local data paths to the critical data path. This, in effect, increases the total time that a given critical data path has to accomplish its function.

2.2.2 Overview of Zero-skew clock design

Circuit speed is a major consideration in high performance VLSI designs. In synchronous digital systems, data are synchronized by clock signal. Hence the performance of the circuit is highly dependant on the clock rate. According to (2.9), the minimum clock period must be greater than the clock skew. As VLSI feature size is scaling down to sub-micron, T_L, T_{Logic} and T_S in (2.10) are decreased significantly. Therefore clock skew can be a bottleneck in increasing the operating clock frequency. Zero-skew clock is desirable choice in any case because it always assures that system operates reliably under maximum permissible clock frequency.

H-tree structures [44] are mostly the most widely used, especially in systolic array designs. H-tree is effective if the clock pins are distributed on a $2^k \times 2^k$ array [37]. H-tree is roughly balanced so that its clock skew is acceptable in most cases. However, H-tree focuses only on wire length balancing, rather than the real objective of balancing clock delay. It is not effective enough for tight skew optimization, as encountered in many high-performance designs nowadays.

The algorithm call *MMM* (Methods of Means and Medians) keeps splitting the area by medians of x or y coordinate of clock pins until thers is only one pin in each sub-region. It still reduces clock skew by balancing wire length. This approach is not affective enough when uneven loads exist [38].

An exact zero-skew clock routing technique that balances the clock delays directly was proposed by *Tsay* [39]. The model of buffered RC tree is obtained by replacing buffer in its equivalent model, as shown in Figure 2.7.

Figure 2.7 Equivalent model for buffer (a) A clock buffer (b) An equivalent model d_b : buffer internal delay c_b : buffer input capacitance R_b : buffer output driving resistance

A lumped delay model is applied on buffered RC subtree. The lumped delay model is derived from

$$t_{kj} = d_i + r_i C_i + t_{ij} \qquad (2.12)$$

where node i is an immediate successor of k, j is a leaf node, d_i is the delay at node i, r_i is the branch delay, C_i is the subtree capacitance, and t_{ij} is the delay from subtree root i to leaf node j.

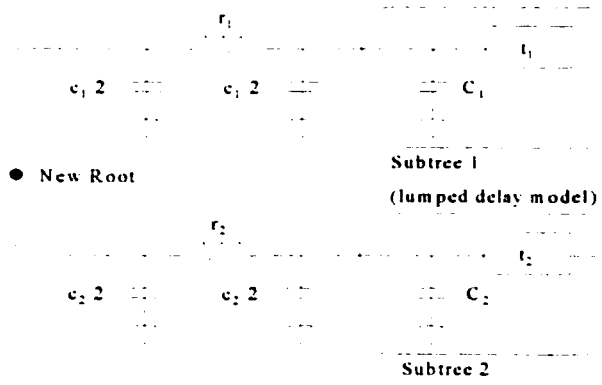

Figure 2.8 Zero-skew merge of two subtrees

Based on the lumped delay model, the algorithm recursively merges two zero-skew subtree to form a new zero-skew tree. It is obvious that the subtree containing only one leaf node is zero-skew

subtree. Hence, leaf nodes are the starting subtrees of the algorithm. The new zero-skew tree means that the delays from its root to all leaf nodes in the 2 subtrees are equal. The location of the root of merged tree is given by (2.13). π model is selected for distributed RC line. Lumped delay model and merging two zero-skew subtrees are shown in Figure 2.8.

$$r_1(c_1/2+C_1)+t_1 = r_2(c_2/2+C_2)+t_2 \qquad (2.13)$$

There are some other algorithms that minimize delay or wire length of clock network during constructing zero-skew clock network [31, 40].

2.3 Low power clock techniques

In a synchronous system, clock network is the fastest and most heavily loaded net. Power dissipation of the clock network is a large portion (20%-50%) of the total power consumption. This is because the clock frequency is typically several times higher than other signals, such as data and control signals.

2.3.1 Clock gating

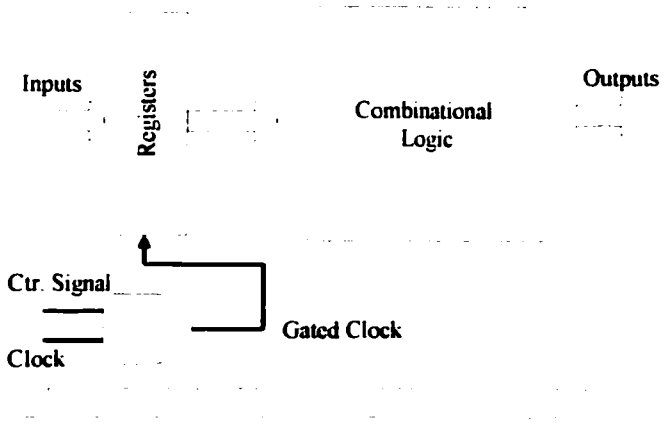

Figure 2.9 Clock gating *vs.* power saving

The technique of clock gating is probably the most well-known and most commonly used form of power management. Clock gating aims at reducing the power dissipation of the functional units as well as that of clock network itself. Clock gating can result in saving due to reduced capacitive switching in the clock network, that may include clock buffers, the interconnect of the clock network, and the latches/registers that are driven by the clock signal. In addition, clock gating disable the new value at the output of the registers, thus saving power consumption in combinational logic fed by the registers due to less switching activity. Figure 2.9 illustrates mechanism of clock gating.

2.3.2 Multiple non-overlapping clocks

Multiple non-overlapping clocks can be used to reduce power dissipation [41]. A circuit driven by single clock has physical capacitance C and switches at clock frequency f. Suppose the circuit is partitioned into two parts, which have physical capacitance $C1$, $C2$, and are driven the two non-overlapping clocks of frequency $f2$ respectively. The power consumption can be saved if the following condition holds.

$$C1 \cdot \frac{f}{2} + C2 \cdot \frac{f}{2} < C \cdot f \qquad (2.14)$$

(2.14) means that as long as $C1 + C2 < 2C$ holds, the use of multiple clock leads to power savings. This condition is likely to hold if the circuit is partitioned carefully so that the registers that are driven by each clock cluster in the layout. Figure 2.10 shows the non-overlapping clocks.

Figure 2.10 Multiple non-overlapping clocks

It is important to note that an operation scheduled in *Clock 1* cannot share a functional unit with an operation scheduled in *Clock 2*. A variable in *Clock 1* cannot share a register with a variable in *Clock 2* either. This technique reduces the power dissipation of the clock network as well as functional units because of reduced switched capacitance. Although the effective clock frequency is still f, the performance of design could degrade due to unavailability of concurrent operations and excessive slack for the operation with short delay. The reason of degrading performance is explained in detail in chapter 4.

2.3.3 Clock distribution using multiple voltages

A clock scheme with multiple supply voltage was proposed in [42] and illustrated in Figure 2.11. In this figure, an HLconverter (High Low converter) is a buffer that converts incoming clock signal from high voltage swing to low voltage swing. The clock signal is then transmitted on the chip as a low voltage signal which is buffered by LLconverters (Low Low converter). The LLconverter can be as simple as inverter, or may be more complex. At the registers where clock pins locate, LHconverters (Low High converter) converts the low voltage clock signal to high voltage signal. In order to save the maximum power, the low power region must be maximized, thus minimizing the high power region. An HLconverter is inserted at the root of the clock tree, and LHconverters are inserted at the clock sinks.

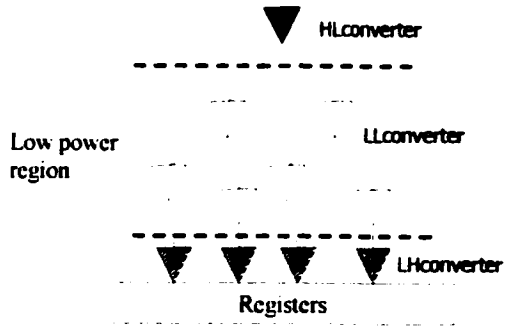

Figure 2.11 Clock with multiple voltages

2.3.4 Minimum SC (switched capacitance) heuristic

The nearest-neighbor heuristic of [43] greedily merges two nodes when the geometric distance between the corresponding merging sectors is at minimum. The algorithm of *Minimum SC* is also greedy, but the merging sequence is determined by the switched capacitance of the merging sector. The merging sector is computed by *Deferred-Merge Embedding* (DME) proposed by [40] to assure that clock tree is zero-skew and its wire length is minimized. The merging sector with small switched capacitance is merged into clock tree first.

Minimum SC not only considers the geometric distance between two nodes to be merged, but also takes the activity of merged sector into account. This means that the high activity nodes tend to merge into the clock tree as late as possible so that overall activity in the tree is reduced. The method is similar to the tree construction of *Huffman* encoding except the method considers the geometry of the nodes too. *Minimum SC* heuristic is illustrated in Figure 2.12.

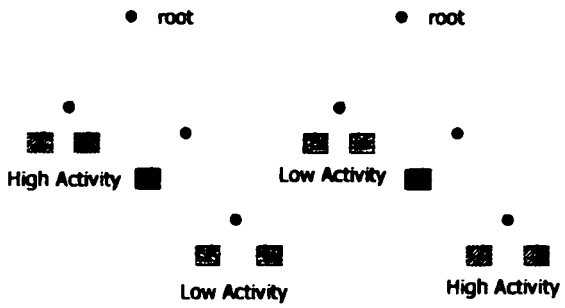

Figure 2.12 Two clock-tree topologies (a) Low activity nodes are merged first

(b) High activity nodes are merged first

CHAPTER 3
ACTIVITY-SENSITIVE CLOCK DESIGN FOR LOW POWER

In this chapter, an activity-sensitive clock tree construction technique is proposed for low power. The chapter first introduces the term of *node difference* to quantitate the similarity of two activity pattern. A rule for control signal with minimum transitions is also suggested. A binary clock tree is built using the node difference between different modules to optimize the power consumption due to interconnections (i.e., clock gating signals and clock edges). After the clock tree is constructed, the gating signals are also optimized to further reduce the power consumption. This algorithm extracts behavior information from CDFG, so it is high-level low power technique.

3.1 Clock Activity *vs* power dissipation

Let $G = \{V, E\}$ denote the clock tree, where $V = \{v_i \mid i = 1, 2, \ldots, m_v\}$ is the set of nodes, and $E = \{e_j \mid j = 1, 2, \ldots, m_v - 1\}$ is the set of clock edges corresponding to each node (except the root node). We use $S = \{v_k \mid k = 1, 2, \ldots, m_s\}$ (where $m_s < m_v$) to denote the set of modules (namely, the leaf nodes). The rest $(m_v - m_s)$ nodes are called internal nodes. The root is said to be at level 0. Node v_i is said to be at level n_i if there are n_i edges on the path from v_i to the root. For each module, we can obtain the activity information from the behavioral level description CDFG which is derived from hardware description language such as VHDL. The activity pattern is a set of '1' and '0', where '1' represents the active period while '0' represents the idle period. For the binary clock tree, a parent node must be active whenever its left child or right child node is active. This means the activity pattern of the parent node is formed by OR-ing the activity patterns of its left child and right child nodes

Activity-driven clock design was first proposed by *Farrahi et al* [44]. Binary tree is adopted as the topology of clock tree. Clock gating is applied on each node except the root of clock tree. The

power dissipation can be saved by combining nodes with similar activity pattern instead of random combination. An example is given below.

Module	Control Step						Activity Pattern
	C1	C2	C3	C4	C5	C6	
M1	X	X	X	X			111100
M2	X	X		X	X		110110
M3	X	X					110000
M4	X	X					110000
A1			X				001000
A2							001000
S1					X	X	000011
C1				X			000100

Table 3.1 After scheduling and resource allocation of the CDFG of *Differential Equation* (DE). This scheduling uses 6 control steps. Resource includes 4 multipliers (M1, M2, M3, M4), 2 Adders (A1, A2), 1 Subtractor (S1), and 1 Comparator (C1).

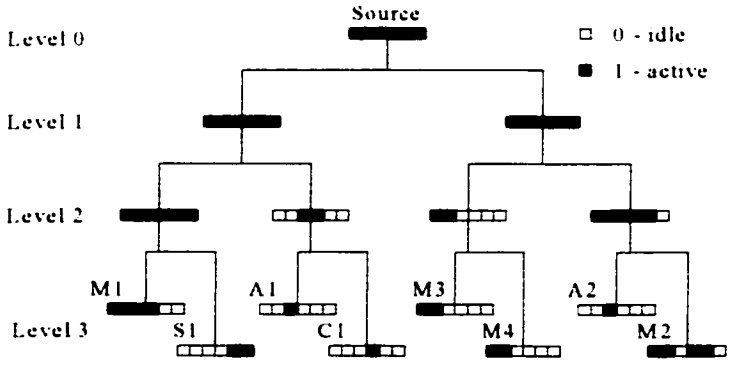

Figure 3.1 DE Example 1

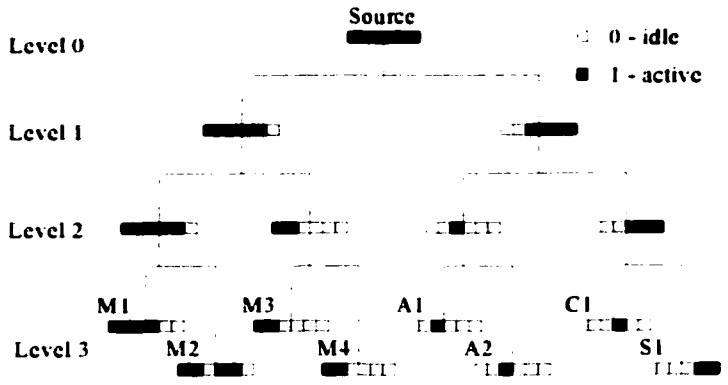

Figure 3.2 DE Example 2

Tree Level	Number of idle periods	
	DE example 1	DE example 2
Level 3	31	31
Level 2	9	13
Level 1	0	3
Power Saving	40/90=44%	47/90=52%

Table 3.2 Comparison of the two clock trees of Figure 3.1 and Figure 3.2 in terms of total number of idle periods

There are totally 90 periods in both examples. In Figure 3.2, two nodes including modules and internal nodes with similar activity pattern are merged to form a parent node. As shown in table 3.2, the total number of idle periods is larger in Figure 3.2 than that in Figure 3.1. Consequently, the power dissipation of the clock network is reduced.

3.2 Node difference based power analysis

Since clock gating implies that additional control signal is need, there exist trade-off between the power dissipation of the control signals and the power saving on the clock edges due to clock gating. The control signals also need extra routing area. The power consumed by the clock at a

frequency f is $\alpha f V_{dd}^2 C_T$, where α is the average number of active periods per clock cycle ($\alpha = 1$ for the ungated clock trees), V_{dd} is the supply voltage, and C_T is the load capacitance of the clock tree. The power dissipation due to control signals is $\frac{1}{2} t_c V_{dd}^2 C_s$, where t_c is the average number of transitions per second and C_s is the capacitance of control signals. In the following discussion, the power dissipation of control signals is taken into account. We also develop a method to determine the control signals with minimum transitions.

3.2.1 Rule for control signal

An example of clock tree is shown in Figure 3.3, where the tree is a binary tree for which each parent node has no more than two children. The control signal gates the parent node to reach the child node. For instance, v_1 is the left child of v_5 which is gated by v_1's control signal $Ctrl$. the activity pattern of the parent node is formed by OR-ing the activity patterns of its left child and right child nodes. In contrast, the activity pattern of a child node is obtained by AND-ing the activity patterns of its parent node and control signal. A general rule for determining the control signals from the activity patterns is given in table 3.3, where the entries "1" (or "0") represent "active" (or "idle") for nodes, and "high" (or "low") for control signals.

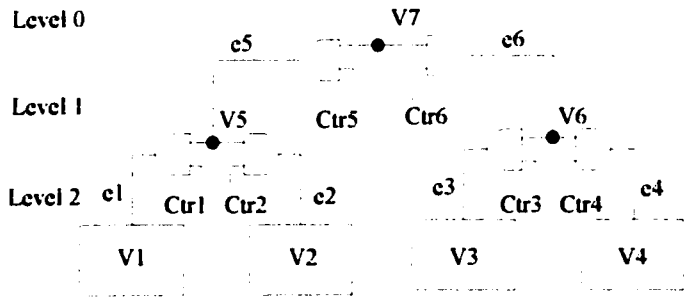

Figure 3.3 A binary gated clock tree

Parent	Child	Control signal
1	1	1
1	0	0
0	0	Unchanged from last control
0	1	(Impossible combination)

Table 3.3 Rule for control signal

An example of determining the control signals is shown in Figure 3.4, where child 1 has the activity pattern {0011100011} (note that each bit corresponds to a control step or period), and child 2 has the activity pattern {0001100110}. By OR-ing operation, we obtain the parent's activity pattern {0011100111}. The control signals in Figure 3.4 are determined using table 3.3. By XOR-ing the activity patterns of children 1 and 2, we have {0010000101}, indicating that there are three bits for which they have different logic values. The number of bits for which the modules have different logic values is referred to as *node difference*. When the parent node and its child node both are logically zero, it does matter whether control signal is 1 or 0. In order to avoid unnecessary transition, the control signal maintains the value at last control step.

Figure 3.4 Activity pattern pf parent node and control signals

3.2.2 Power consumption under different configurations

Node difference is proposed to measure how the activities of two nodes are similar. The following example indicates that not only the number of idle periods of the clock edges, but also the number of transitions of control signals is also related to node difference.

Consider an example with 2 multipliers (M1 and M3) and 2 adders (M2 and M4) with their activity patterns shown in table 3.4. We assume: (i) For each active period, the multiplier and adder contribute 8 units and 4 units, respectively, to the total power consumption. They consume no power during the idle periods. (ii) The root of clock tree is located in the central. Each control signal is generated from the central. (iii) The control signal is routed along the clock edges. The other end of child's control signal is placed as close as possible to the parent node. In Figure 3.5 and Figure 3.6, the wire length of *Ctr5* or *Ctr6* is 0, i.e., Ctr5 or Ctr6 consumes no power in spite of transitions over them. The root has no control signal. (iv) For each active period, the clock edge at each level (except level 0) of the clock tree contributes *2* units to the total power. (v) Every *8* transitions of a control signal at each level 2 contribute *1* unit to the total power.

Figure 3.5 and Figure 3.6 show two possible clock tree topologic structures/configurations for these four modules. Both configurations have three levels with two internal nodes, M5 and M6, at level 2. The difference is that in Figure 3.5, the modules (i.e., nodes) with similar activity pattern (i.e., with smaller node difference) are combined first. By OR-ing operation, we can obtain the activity patterns of M5 and M6 from their children. The control signals (i.e., *Ctr1* through *Ctr6*) can be obtained using table 3.3. Table 3.5 summarizes the comparison of these two configurations. Also included in table 3.6 is the ungated configuration without control signals, as shown in column "Ungated".

It can be seen from table 3.6 that while both configurations 1 and 2 save power consumption by clock gating, configuration 1 is much better than configuration 2 which consumes more power due to clock edges and control signals. This is because configuration 1 considers the node difference while configuration 2 does not. As shown in table 3.5, the number of transitions of control signals is likely to be less when node difference between two nodes is reduced.

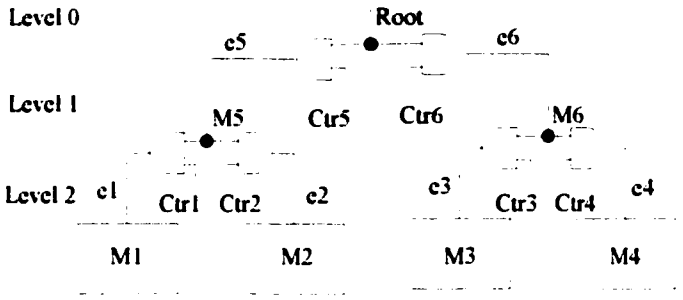

Figure 3.5 Configuration 1

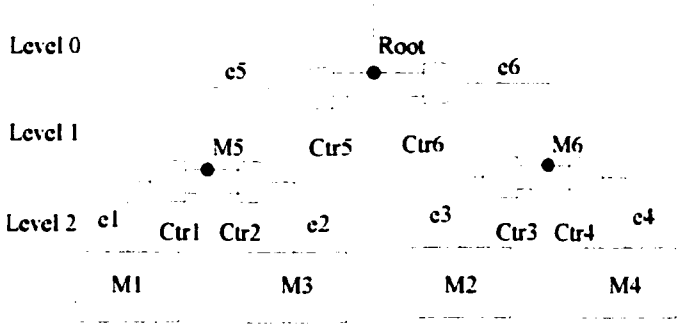

Figure 3.6 Configuration 2

Module	*Activity pattern*
M1 (multiplier)	10101010
M2 (adder)	10101010
M3 (multiplier)	01010101
M4 (adder)	01010101

Table 3.4 Modules and their activity patterns

Item	*Config. 1*	*Config. 2*
M5	10101010	11111111
M6	01010101	11111111
Root	11111111	11111111
Ctr. 1	11111111	10101010
Ctr. 2	11111111	01010101
Ctr. 3	11111111	10101010
Ctr. 4	11111111	01010101
Ctr. 5	10101010	11111111
Ctr. 6	01010101	11111111

Table 3.5 Activity patterns of internal nodes and control signals

Item	Config. 1	Config. 2	Ungated
Active periods at level 1	8	16	16
Active periods at level 2	16	16	32
Transitions at level 2	0	32	0
Power for modules	96	96	192
Total power	144	174	298

Table 3.6 Power consumption for different configurations

3.2.3 Power analysis

In this section, we will show that node difference plays an important role in low-power clock tree construction. For an extreme case where two nodes with the same activity pattern are combined, the activity pattern of parent node is the same as its children's activity patterns. Thus, no transitions

occur with the control signals. The number of active periods for the parent node is the same as that for each child. We refer to this extreme case as the *ideal point*. If we change one child's distribution of activity periods while keeping the number of its active periods unchanged, the node difference between two children is increased by exactly the increased number of active periods of the parent node. This is given by the following equation:

$$A_{inc} = L_d = (A_p - A_{ch1}) + (A_p - A_{ch2}) \tag{3.1}$$

where A_{inc} is the increased number of active periods, L_d is the node difference between child 1 and child 2, and A_p, A_{ch1} and A_{ch2} are the numbers of active periods for the parent, child 1 and child 2 nodes, respectively. Since reducing the A_{inc} can lead to the saved power of the clock edges, the nodes with smaller node difference should be combined with higher priority during the clock tree construction.

An intuitive idea is that minimizing the node difference of neighboring nodes can result in the reduced number of control signals' transitions required. Since the state of a control signal depends on its previous state when both parent and children are idle (refer to table 3.1), it would be impossible to formulate the relationship between the node difference and transitions of control signals by combinational logic. Therefore, we conducted some experiments to obtain the correlation between the total transitions of control signals and total node difference of all modules (i.e., nodes). Figure 3.6 shows the experimental result with 20 nodes and 20 clock periods. In this experiment, we keep the activity densities for all nodes fixed, and the nodes are paired sequentially (i.e., node 1 is paired with node 2, and node 3 is paired with node 4, etc.). The total transition is the sum of transitions of control signals for all nodes. The total node difference is the sum of node differences for all node pairs. We change the total logic distance by shuffling the activity patterns of nodes without changing the activity density, and obtain the corresponding total transitions. It can be seen from Figure 3.6 that the number of total transitions of control signals is proportional to the total node difference.

In order to take into account the power consumption for both clock edges and control signals during the clock tree construction, we define a *merging power*, for each pair of modules to be

combined, as a weighted sum of logic distance (i.e., L_d) and the number of transitions of control signals (denoted by T_c):

$$P_{merg} = W_{clk} * L_d + W_{ctr} * T_c \qquad (3.2)$$

where W_{clk} and W_{ctr} are the weights for clock edges and for control signals, representing the power consumption contributed by each active period of clock edges and by each transition of control signals, respectively. These weights are proportional to the wire length of the associated clock edges or control logic (The computation of wire length is shown in equation 3.3-3.6).

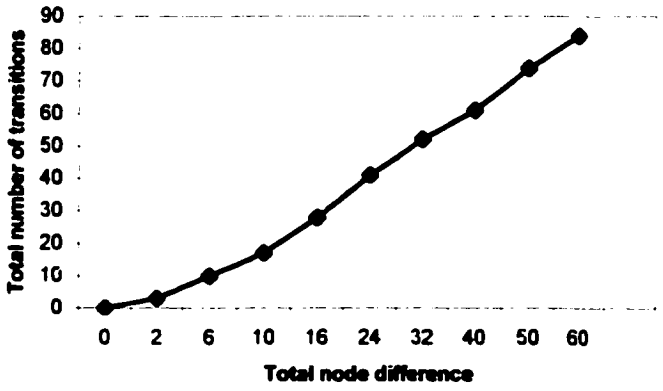

Figure 3.7 The correlation between total transitions and total node difference

3.2.4 Power consumption of binary clock tree

Consider a typical construction process of the binary clock tree. The top level of the tree is the clock source. Leaves or modules are at the bottom level. Each node has no more than two children nodes. The number of nodes at each level is not necessarily power of 2, depending on the total number of modules. If, at a specific level, the number of nodes is odd, there must be one node that doesn't combine with any other node.

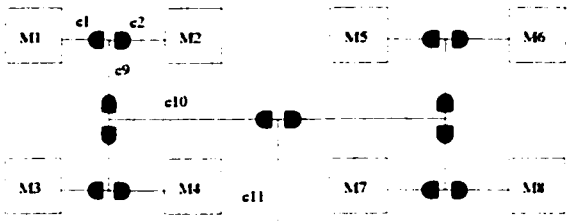

Figure 3.8 The topology of H tree

To estimate the physical wiring information, we take an H tree for example. Figure 3.7 shows the H tree with eight modules that are placed in uniform grids. Assuming the clock wiring length at the bottom level is l_{clk}, the clock wiring length at level n is given by

$$l_{clk(0)} = 0 \tag{3.3}$$

$$l_{clk(n)} = 2^{fl[(N-1-n)/2]} l_{clk} \tag{3.4}$$

where N is the number of levels ($0 \le n < N$), and $fl[x]$ is the floor value of positive number x. Wire length of the source is 0. Equation 3.4 means that the wire length of clock edges doubles every other level upward. Assuming all control signals are generated from the central [43]. The routing of the control signals is parallel to the clock routing.

The wire length of each control signal at level n is given by

$$l_{ctr(n)} = 0 \qquad for\ n = 0,1 \tag{3.5}$$

$$l_{ctr(n)} = l_{clk} \sum_{k=1}^{n-1} 2^{fl[(N-1-k)/2]} \quad for\ n > 1 \tag{3.6}$$

Since there is no control signal for the source, $L_{ctr(0)}$ is 0. If we assume the clock gate is placed as close as possible to the upper level, then $L_{ctr(1)}$ can also be considered as 0. As to other levels, the wire length of control signals is the sum of the wire lengths of upper level clock edge all ways up to the root. Therefore, the power consumed by clock edge i at level n can be expressed as:

$$P_{clk(i)} = k_{clk} A_i L_{clk(n)} = W_{clk(n)} A_i \qquad (3.7)$$

where k_{clk} is a constant, $W_{clk(n)} = k_{clk} L_{clk(n)}$, and A_i is the number of active periods of the node connected to clock edge i. Similarly, the power consumed by control signal j at level n is given by:

$$P_{ctr(j)} = k_{ctr} T_j L_{ctr(n)} = W_{ctr(n)} T_j \qquad (3.8)$$

where k_{ctr} is a constant, $W_{ctr(n)} = k_{ctr} L_{ctr(n)}$, and T_j is the number of transitions of control signal j. The total power consumption, P, of the clock network consists of the contributions by modules (P_m) and by interconnections, i.e.,

$$P = \sum P_m + \sum P_{clk(i)} + \sum P_{ctr(j)} \qquad (3.9)$$

3.3 Algorithms

Based on the above discussions, we develop a so-called Merging Algorithm (MA) for the low-power clock tree construction. The algorithm consists of two parts. Given the total number of clock periods, the number of modules and their activity patterns, we first construct the binary clock tree based on (3.6) in a bottom-up manner. This is done level by level toward the root node of the tree. Then we perform an optimization step by looking at the tradeoff between the power penalty from control signals and the power savings from gated clock edges.

3.3.1 Clock Tree Construction

The basic idea for the clock tree construction is to combine every two nodes with smallest value of P_{merg} from leaf nodes towards the root node. All node pairs at one level are combined to obtain their parent nodes at the next level. If the number of nodes at a specific level is odd, then one node at this level will combine with a dummy node that has no active periods and no transition on its control signal. Each parent's activity pattern is obtained by OR-ing the activity patterns of the children. The control signals of children are determined by the method described in Section 2.

From (3.2), the *merging power* for node i and j at level n is given by

$$P_{merg(i,j)} = W_{clk(n)} L_d + W_{ctr(n)} (T_i + T_j) \qquad (3.10)$$

where T_i (T_j) is the number of transitions of the control signal for node i (node j). If there are M nodes at a given level, the number of possible *merging powers* values is $M(M-1)/2$. We sort all these values in ascending order. The two nodes with smallest value are paired to form their parent node, and then they are deleted from the list. This process repeats until all nodes are paired. If the number of nodes is odd, the last node pairs with a dummy node. All parent nodes so obtained at the upper level can be used to determine the control signals of their children. The depth of the tree is *ceil*(log m_s) — the closest integer that is larger than m_s, where m_s is the number of modules. This is a greedy algorithm without ensuring that the total interconnection power is minimum.

3.3.2 Local ungating

After the clock tree is constructed, every node (except the root) has an associated control signal. The control signals can reduce the power consumption due to clock edges and/or modules. However, the transition of control signals consumes additional power, which may offset the power savings. This requires more attention especially at some levels close to the leaf nodes, where the weights of control signals are relatively high (see (3.6)). The control signals' power penalty caused by their transitions can be reduced by incrementally changing some of their periods from '0' to '1'.

This operation is called *local ungating*, which is acceptable if it can result in the overall power reduction.

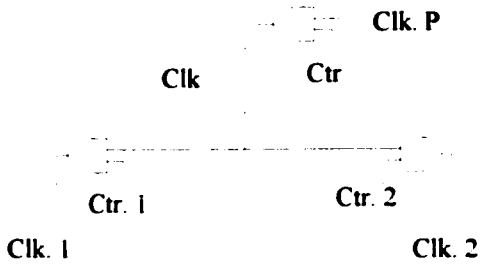

Figure 3.9 Evaluation of local ungating

input: *initial binary tree*
output: *optimized tree with updated control signals*

begin
 for *each node in the tree* ***do***
 Find each dent of its control signal's waveform, and do a tentative change from '0' to '1' for control signal (ctr);
 Update the activity pattern (clk) of the node, and reshape the control signals (ctr1 and ctr2) of its children locally;
 if *the node is at the bottom level* ***then***
 $\{ \Delta P \leftarrow \Delta P_{clk} \cdot \Delta P_{ctr} \cdot \Delta P_m \}$
 else $\{\Delta P \leftarrow \Delta P_{clk} \cdot \Delta P_{ctr} \cdot \Delta P_{ctr1} \cdot \Delta P_{ctr2}\}$
 if $(\Delta P > 0)$ ***then***
 the local change is accepted with updated control signals;
 end

Table 3.7 Power optimization by local ungating

Figure 3.8 illustrates an example. Suppose initially *Ctr* = {11101110} which is to be changed into {11111110}. To keep the effect to be local, we can reshape the signals Ctr.1 and Ctr.2 such that Clk.P, Clk.1, Clk.2 and other nodes across the tree remain unchanged. Thus, we only need to evaluate the power consumption due to changed control signals locally. If the power value is reduced, the local change is accepted. The similar process can be applied to each node from the bottom level to the root. For the nodes at the bottom level, the power variation of the clock edges, control signals and modules is involved in the evaluation. For the nodes at other levels, the power evaluation is performed for their clock edges, control signals and their children's control signals. The procedure of power optimization by local ungating is shown in table 3.7.

3.4 Experimental results

We created the *CDFG* (Control Data Flow Graph) files from the benchmarks described in VHDL by using *CDFG Tool, version 1.0.* [21]. The ASAP (i.e., *As Soon As Possible*) scheduling was done based on the CDFG files. After the scheduling, we obtained the number of periods and minimum resource required, i.e., modules that include adder, substractor, multiplier and multiplexer etc. The activity patterns of modules are thereby acquired by resource allocation that assigns operations of each period to the relevant modules. In our tests, the power consumption of adder, substractor and multiplexer is assumed 4, while that of multiplier, divider is 8.

The merging algorithm is applied on the activity patterns followed by local ungating. We compared the merging algorithm with conventional clock construction that does not account for node difference. The results are shown in table 3.8. The merging algorithm results in fewer active periods on clock tree and fewer transitions of control signals, reducing power consumption of the clock tree. While local ungating on the bottom level may lead to increase of module power, the overall power consumption is saved. By combining the nodes with smallest node difference, it is much likely to produce the control signals that are always high at most periods. Thus, the ungating reduces the number of control signals and total wire length as well. As can be seen from table 3.8, there is a significant decrease of the control signals' wire length.

Bench-mark	# Periods	# Modules	Power savings	Wire-length reduction
iir7	13	19	13.5%	50.9%
ellipf	14	8	21.2%	30.9%
diffeq	5	8	23.1%	57.1%
parallel	9	17	24.8%	73.9%
nc	12	26	28.9%	55.0%

Table 3.8 Performance of merging algorithm on benchmarks

3.5 Conclusions

In this chapter we have dealt with the activity-sensitive clock tree construction for low power. The method of determining the control signals is presented such that their transitions are reduced. While the clock tree construction algorithm is based on *node difference*, the local ungating technique is to shape the control signals so that power consumption can be reduced further. Experiments show that the power savings are strongly related to modules' activity information, and that node difference plays an important role in low-power clock tree design.

CHAPTER 4
LOW POWER CLOCK BASED ON CLOCK FREQUENCY REDUCTION

This chapter proposes a high-level power optimization scheme with two techniques: operator chaining, multiple clocks. In operator chaining, primitive operators with shorter delays are chained to create a complex operator with longer delay. Then a lower clock frequency can be applied without increasing slack. In multiple clocks, the operators with longer delays can be driven by another clock with lower frequency while other operators are still driven by the original clock. By wholly or partially reducing clock frequency, the power dissipation by clock is decreased because of less switched capacitance. Since both techniques avoid increasing slack, they have no or little effect on the performance of circuits.

4.1 Introduction

Clock power is dissipated on charging and discharging of the load capacitance, as shown in the following golden equation [26]:

$$P = \alpha f C_L V_{dd}^2 \qquad (4.1)$$

where f, C_L and V_{dd} are the clock frequency, capacitive loading and supply voltage, respectively, and α represents the active ratio which is the percentage of active clock cycles over total clock cycles per second when clock gating is enabled. From this equation, reducing V_{dd} is the most effective way to save power. For instance, one can use multiple supply voltages and a reduced swing to achieve low power clock [42, 45] while supply voltage reduction lead to delay increase of operators. Hence, supply voltage reduction is usually only applied on non-critical paths. In order to reduce the capacitance C_L, both the wire length and the number of inserted buffers in the clock network need to be minimized [40, 46]. When the clock edges with similar activity pattern are merged, the clock can be shut down at upper level of the clock tree during their idle periods [44].

Physical constraints also are considered during the gated clock tree design [47]. This reduces the active ratio α in the above equation, and hence results in power savings. Both wire length and activity are taken into account to reduce switched capacitance [43]. Also, when the timing performance permits, the clock frequency can be kept as low as possible for power reduction of clock signals.

In this chapter, we present an integrated optimization scheme at high level to achieve a low-power clock tree. We target towards reducing the activity of the whole or part of a clock tree, while keeping the system latency or performance.

The chapter is organized as follows. In section 2, we describe the operator chaining and show that a lower clock frequency can have only limited effect on the system latency/performance. In section 3, we propose multi-clock method where the operators with longer delay are driven by another clock with lower frequency. Section 4 reports experimental results, and Section 5 concludes the schemes.

4.2 Operator chaining scheme

4.2.1 Operator chaining

Operator chaining is to combine two or more operators into a single operator. This is realized at register-transfer level by connecting the output of a front operator directly to the input of the next operator (i.e., the registers at the output stage of the front operator are removed). Figure 4.1 illustrates operator chaining in a RT level design.

The delay of chained operator is less than the sum of involved single operators. The reason is that the delay of intermediate register is deducted. Therefore, consecutive operations can be carried out by corresponding chained operator with less delay. Operator chaining normally results in bigger area size that is also a major concern on VLSI design.

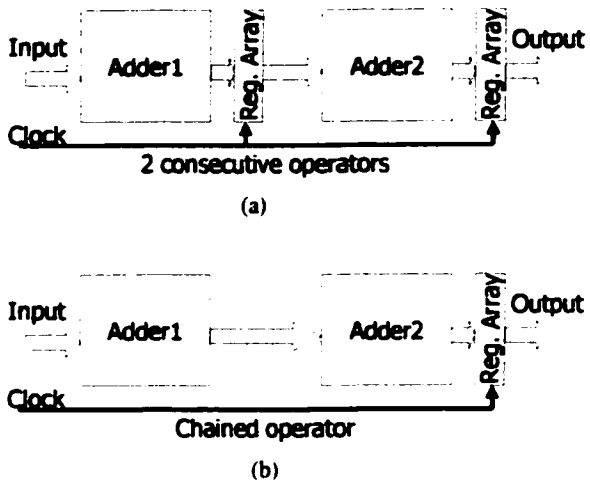

Figure 4.1 Chaining of two adders

Figure 4.2 shows an example of scheduling before and after chaining performed on two adders. It is assumed in Figure 4.1(a) that multiplication and addition are mapped to a multiplier with 100ns delay and an adder with 50ns delay. The clock cycle is selected at 50ns, and the multiplication is completed over two clock cycles. Two adders in Figure 4.2(a) are chained into a chained operator in Figure 4.2(b). The chained operator normally has a delay less than double of the adder delay (i.e., 100ns). If the clock cycle is set to 100ns, both Figure 4.2(a) and (b) have the same latency of 100ns.

Table 4.1 shows some operators' data with 0.35 μm process provided by CMC (Canadian Microelectronics Corp). In this table, *Addadd* denotes a chained operator which is made up of two adders, *Addsub* is a chained operator formed by chaining an adder and a subtractor, and *Subsub* is a chained operator consisting of two subtractors. It is noticed that the delay of a chained operator is less than the sum of delays of its member operators, and that the chained operator inherits all the inputs and outputs of its member operators. Figure 4.3 shows another example of operator chaining, where there are 3 inputs and 2 outputs before chaining (see Figure 4.3(a)) and the chained operator has the same inputs and outputs (see Figure 4.3(b)).

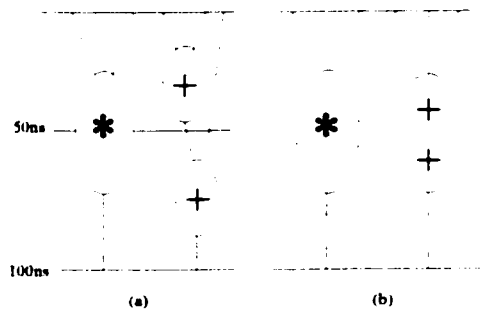

Figure 4.2 Operator chaining on two adders

Operator (16 bits)	Delay (*ns*)	Energy (*pJ*)	Area (μm^2)
Adder	9.8	57	10255
Subs	9.8	57	10832
Mul	21.5	622	124652
Divider	21.5	622	124652
Addadd	10.9	119	20510
Addsub	10.9	120	21087
Subsub	10.9	122	21665

Table 4.1 Data of some operators built from 0.35 μm process

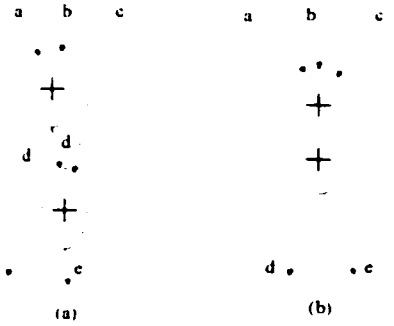

Figure 4.3 I/Os of the chained operator

4.2.2 Multicycling

Clock selection has great impact on the performance of the VLSI system. Figure 4.4 shows the difference of performance between the clock period at 100ns and at 50ns. The multiplication is executed over two control steps, as shown in Figure 4.4 (b). We refer to an operation such as this multiplication as a *multicycle operation* – it will have to execute continuously throughout its entire execution time, and its input values must be latched throughout that period as well [22]. In this example, multicycling can decrease the length of schedule to 100ns. However, multicycling uses two or more control steps, which may result in a larger controller and extra control signals. Figure 4.5 gives a RT level implementation of multicycling.

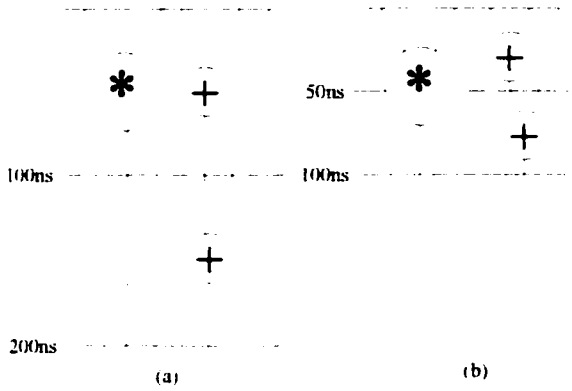

Figure 4.4 Multicycling in scheduling

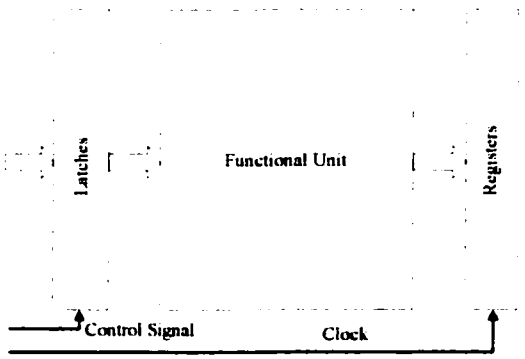

Figure 4.5 RT level implementation of multicycling

4.2.3 Operation mapping

Mapping refers to replacing groups of primitive hardware units with more complex units such as chained operators [48]. Mapping can be divided into *complete* and *partial* matches. Figure 4.5 is an example of complete match. In a CDFG which describes operations, control signals and their relationship, if one addition is followed by another addition, then we can execute the two additions with an Addadd. Alternatively, we can execute a single addition with an Addadd whose third input is set to 0. The latter is called the partial match as illustrated in Figure 4.6.

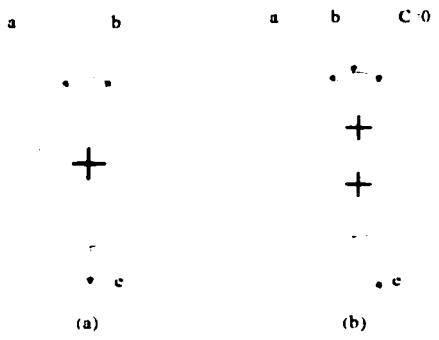

Figure 4.6 Partial match

We go through CDFG by detecting the first member operation of the chained operations. If the operation and its following operations completely match the chained operation, we carry out a replacement due to complete match. After all complete matches are processed, we then conduct replacements due to partial match by finding the potential operation that is a member operation of the chained operation but can not be mapped into any available primitive operator. For example, in a resource pool where there is an Addsub but no Subs operator, any subtraction in a given CDFG should be replaced by the Addsub. Note that we are only interested in the resource constrained scheduling (RCS) in which the type and number of operators are limited. If an operation is involved in more than one chaining, we pre-define the priority in terms of replacement according to power consumption. The chained operator with less power consumption is given higher priority in the matching list.

4.2.4 Power savings

Power consumption in clock can be reduced by selecting a lower clock frequency and performing the RCS on CDFGs. However, power consumption in module may increase due to the partial matches. For the CDFG where complete matches dominates, the power savings in clock would be significant compared with the power penalty in module. In terms of performance, the partial match tends to increase the total delay while the complete match can reduce the total delay with careful selection of the clock frequency. In general, the performance can be maintained in the complete-match dominant CDFGs.

Lower clock frequency can also eliminate the multicycling. In Figure 4.4 (b), for instance, the multiplication is executed over 2 clock cycles. This is referred to as a multicycle operation. The input of multicycle operation must be latched throughout the whole execution time [22]. Multicycle also needs an extra control signal, resulting in more area. If the multiplication is completed within one clock cycle instead (as shown in Figure 4.4 (a)), the single-cycle multiplier occupies less area than multicycle multiplier does. In this sense, the chaining scheme can lead to a smaller chip size.

4.3 Multiple clocks

Multiple clocks means that there are a few separate clocks, each of which drives the functional units in different region. Those clocks may differ in frequency, or phase. The power dissipation can be reduced by applying the frequency lower than that of original single clock on the clocks.

4.3.1 Multiple non-overlapping clocks

A multiple non-overlapping clocks scheme is proposed by *Papachristou* et al [41]. The power dissipation of clock can be reduced due to frequency reduction. It assumes that all kinds of operations are completed in one control step, neglecting the difference on time consumption among distinct operations. Figure 4.7 presents the CDFG to be scheduled. Figure 4.8 illustrates the difference on performance between a single clock with the period of 50ns and two non-overlapping

clocks with the period of 100ns. Assume that the delay of multiplication is between 50 ns and 100ns, and the delay of addition is less than 50ns. The resources for both schedulings are one multiplier and one adder.

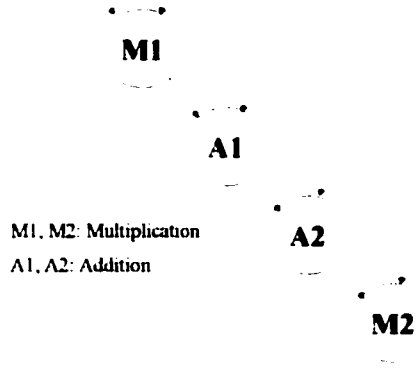

Figure 4.7 An example of CDFG

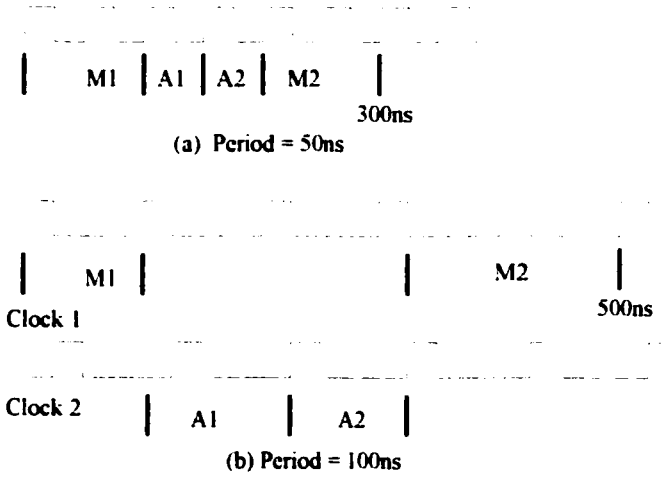

Figure 4.8 Difference between single clock and two non-overlapping clocks

The total delay for Figure 4.8 (b) is 500ns, which much worse than the total delay of Figure 4.7 (a) that is 300ns. The performance of Figure 4.8 (b) is usually unacceptable. As indicated by the waveforms in Figure 4.8, the extra delay is introduced by

1) Extra slack in adder because of longer period of the clock.
2) The two clocks run at different phase. The phase gap is appended on the total delay when there is data exchange between the functional units belonging to different clocks.

In order to achieve better result, the difference between the delays of functional units must be taken into account. It would be better find a way somehow to eliminate the phase difference.

4.3.2 Multiple clocks scheme

The basic idea of multiple-clock is to allow different clock zones with different frequencies, so that part of the chip runs at lower frequency for clock power reduction. The operators with short delay are still clocked at higher frequency. A typical example of multiple clocks is shown in Figure 4.9 where the multipliers have much longer delay than the adders. In particular, all the operators of same type should be located in one clock zone so that no resources in different clock zones can be shared.

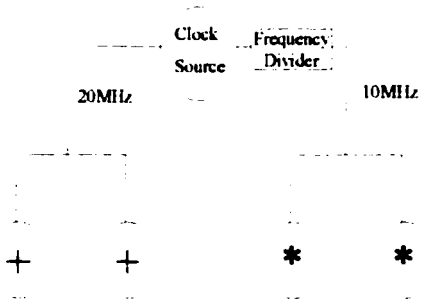

Figure 4.9 Multiple clocks

Also, the starting time of an operation must be a multiple of its driving clock period. For example, if the multiplier's clock is 10MHz, the multiplication can only start at 100ns, 200ns, 300ns, and so on. This restriction may increase the total delay. An example is shown in Figure 4.10 (b) where the adder and multiplier are driven by different clocks. The 20MHz (i.e., 50ns each period) is required for the adder, while the 10MHz (i.e., 100ns each period) is needed for the multiplier. Both adder and multiplier run at 20MHz in Fig. 5a with the total delay of 150ns. An extra delay of 50ns is generated in Figure 4.10 (b) since the multiplication can only be scheduled at the time of a multiple of 100ns. In this work, we assume binary clock scheme where one clock period is a multiple of another clock period. If there is an addition is followed by a multiplication (refer to Figure 4.10 (b)), the multiplication can start right after its preceding addition is completed. The ending time of multiplication is always a multiple of 50ns, which is the period of the clock driving the adder.

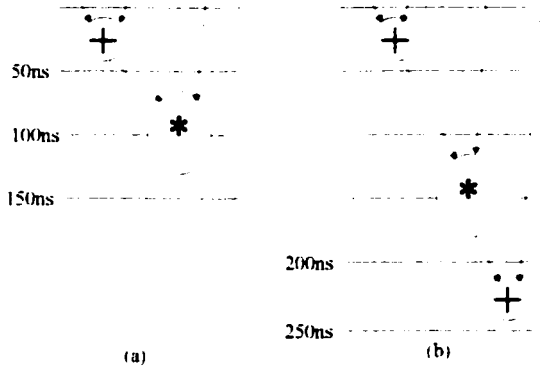

Figure 4.10 Multiple clocks and delay

For those CDFGs that are rich of the operations with long delay (such as multiplication), the multiple-clock scheme reduces the clock power without degrading the performance. The power consumed by modules remains unchanged. To obtain the clock with lower frequency, we need a frequency divider that could be as simple as a flip-flop. Thus, the power and area penalty caused by the frequency divider can be ignored.

4.4 Experimental results

At the behavioral level, no floor planning and placement of the design is available. To estimate the physical wiring information, we assume the clock tree is a H-tree because H-tree is wire length balanced and hence nearly zero-skew. The wire length at level *n* is given by equation 3.3 and 3.4. The wire length of clock edges doubles every other level upward. The topology is shown in Figure 3.8.

Multiple-clock technique divides the clock tree into multiple sub-trees. Inter-tree clock edges integrate the sub-trees into a single clock tree. The wire length of the inter-tree clock edges is given by equations 3.3 and 3.4. Since both the operator chaining and multiple clocks may affect the delay, the latency was normalized in our experiment in order to evaluate the power consumption.

We first generated the CDFG description using the tool offered by [21], and performed the resource constrained ASAP (As Soon As Possible) scheduling with the library shown in table 4.1. The activity of each module was then obtained after scheduling and resource allocation. We carried out the operator chaining or multiple-clock technique on all tested benchmarks, and selected a better result. For the operator chaining, the clock period is selected to be 22ns. For the multiple-clock technique, the clock periods are set to 11ns and 22ns. As the chaining technique may increase the power consumption of the modules, a trade-off was made between the module power penalty and clock power savings. The objective to be optimized is the overall power savings.

Table 4.2 shows the experimental results on eight benchmarks, where *lattice*, *nc* and *iir7* were optimized by multiple-clock technique while the rest were optimized by operator chaining. As can be seen from the table that the clock power reduction ranges from 33% to 57% with an average of 46%. The total power savings are from 11% to 18% with an average of 15%. Meanwhile, the increase, if any, in both the area and latency is negligible for all circuits, as shown in the table.

Benchmark	Before optimization		After optimization		Clock power	Total power
	Area (μm^2)	Latency (ns)	Area (μm^2)	Latency (ns)	Red. rate (%)	Red. rate (%)
lattice	281,591	143	270,391	165	33	11
cascade	281,014	209	269,814	220	51	17
diffeq	285,686	110	285,319	132	48	18
ellipf	151,339	297	155,994	308	57	17
fir11	281,014	143	290,324	154	53	16
nc	291,846	385	270,391	385	38	12
parallel	281,014	242	290,324	264	54	17
iir7	281,591	187	270,391	187	34	11
Average					**46**	**15**

Table 4.2 Optimization results on benchmarks

4.5 Conclusions

In this chapter, a new high-level optimization scheme is proposed to reduce the clock power consumption. Unlike the approaches at logic or physical level [42, 45, 47], our method reduces the power at high level. Operator chaining technique is suitable for the CDFGs that are abundant of complete matches, where the delay and module power consumption can be maintained while reducing the clock power. Multiple-clock technique is effective for the CDFGs where the operations with long delays are dominant. Both techniques can be followed by activity-driven clock design [44] to further reduce the clock power consumption. Since chaining and multiple clocks might introduce slack for the given operators, multiple supply voltages can applied to make use of those slacks without penalty on the performance.

CHAPTER 5
CONCLUSION AND FUTURE WORK

5.1 Conclusion

The portable electronic devices such as cellular phone, laptop, and PDA are usually powered by rechargeable batteries. The power consumption is a very critical parameter in these portable electronic devices. Clock network has highest transition rate in digital VLSI systems, and it accounts for a large portion of total power.

The power consumption of clock can be reduced at behavioral level. CDFG which is derived from HDL description of circuits shows operations, data and control flow of circuits at behavioral level. By scheduling and resource allocation, the activity pattern of each functional unit is obtained. In activity-sensitive clock design, clock gating is applied on the clock edges at each level to save power. It is noticed that two nodes with similar activity pattern result in a parent node with less increased active periods, therefore clock power can be saved by properly pairing nodes at each level. *Node difference* is proposed to measure the difference between the activity patterns of two nodes. Since clock gating leads to additional control signals which also consume power, *merging power* is defined as the sum of the increased power on the parent clock edge and the power of its corresponding control signal to evaluate the cost of combining two children nodes. At each level, all possible *merging powers* are computed and sorted in ascending order. Clock tree is built up in the way which two nodes with smallest *merging power* are paired to obtain their parent node at the upper level until all nodes are merged. Thus, this greedy algorithm saves clock power by reducing the active periods of clock signal and control signal. The following local ungating scheme makes a trade-off between the power penalty on the children clock edges and the power saving on the control signals to optimize the clock power consumption further. In activity-sensitive clock design, both clock power saving and incurred power penalty of control signals are taken into account in low power clock design.

Note that clock power can be reduced linearly by reducing clock frequency. By chaining the primitive operators with short delays, the complex operator with long delay can be created to realize the functions of its member operators. Then the original CDFG is shaped by replacing the consecutive primitive operations with the operation of the complex operator according to complete match or partial match. In the complete match dominant CDFG, the selection of low frequency clock has little or no effect on the performance because the slack of the design has only little change. In multiple clocks scheme, clock power consumption is reduced due to partial clock frequency reduction. Only the operators with long delays are driven by the low frequency clock while the operatiors with short delays still run on the high frequency clock, thereby the effect of splitting clock network on the performance is limited. Multiple-clock scheme is effective for the CDFGs where the operations with long delays are dominant.

In the above techniques, clock power is saved by reducing its switched capacitance or frequency at behavioral level. Thus the estimation of lower bound of clock power consumption can be made on the early stage of design.

5.2 Future work

In activity-sensitive clock design, H-tree is used to estimate physical wire length. H-tree may not satisfy the skew requirement of some synchronous systems and often leads to excessive wire length. The power consumption due to interconnections among modules is not considered either. So an early estimation of floor planning and placement at behavioral level based on the power consumption of interconnections is desired. The power consumption of interconnections could be estimated based on how often the data are exchanged between two modules. Then an exact zero-skew clock algorithm can be used to obtain the accurate wire lengths and load capacitances of clock edges and control signals. Then a zero-skew low power clock is constructed under an accurate model.

Activity-driven clock design can follow operator chaining or multiple clocks scheme to optimize clock power further. Because the period of clock cycle is altered in clock frequency

reduction, it would be possible to exploit the slack by the way of multiple supply voltages. Thereby, the power consumption of modules can be reduced.

APPENDIX (A)

VHDL Source Code of the Benchmarks

//Only two VHDL source codes and their CDFG files are given in appendix A and appendix B //respectively due to limited space. As to other benchmarks used in the thesis, please refer to //http://poppy.snu.ac.kr/~shleee/class/icda2001/CDFGTool.pdf

//Cascade

```
entity cascade is
        port (
                inp: in real;
                outp: out real
        );
end cascade;

architecture cascade_beh of cascade is
begin
        process (inp)
                variable state0, state0_1, state0_2: real;
                variable state1, state1_1, state1_2: real;
                variable state2, state2_1, state2_2: real;
                variable state3, state3_1, state3_2: real;
                variable inp1, inp2, inp3, inp4: real;
        begin
                inp1 := inp * 0.005656462366;
                state0 := inp1 + 0.2855823838 * state0_1 +
                                (-0.9116860618) * state0_2;
                inp2 := state0 + 0.4512591755*state0_1 + state0_2;
                state0_2 := state0_1;
                state0_1 := state0;

                state1 := inp2 + 0.4457342317 * state1_1 +
                                (-0.913078673) * state1_2;
                inp3 := state1 + (-1.09869455)*state1_1 + state1_2;
                state1_2 := state1_1;
                state1_1 := state1;

                state2 := inp3 + 0.196901665 * state2_1 +
                                (-0.9695353354) * state2_2;
                inp4 := state2 + (-0.0099665157)*state2_1 + state2_2;
                state2_2 := state2_1;
                state2_1 := state2;

                state3 := inp4 + 0.5515388641 * state3_1 +
                                (-0.9705899009) * state3_2;
                outp <= state3 + (-0.7304169706)*state3_1 + state3_2;
                state3_2 := state3_1;
                state3_1 := state3;
        end process;
end cascade_beh;

configuration cfg_cascade of cascade is
        for cascade_beh
        end for;
end cfg_cascade;
```

//Differential Equation

```
entity diffeq is
   port (Xinport: in integer;
         Xoutport: out integer;
         DXport: in integer;
         Aport: in integer;
         Yinport: in integer;
         Youtport: out integer;
         Uinport: in integer;
         Uoutport: out integer);
end diffeq;

--VSS: design_style BEHAVIORAL

architecture diffeq of diffeq is

begin

P1 : process (Aport, DXport, Xinport, Yinport, Uinport)
   variable x_var,y_var,u_var, a_var, dx_var: integer ;
   variable x1, y1, t1,t2,t3,t4,t5,t6: integer ;

begin

  x_var := Xinport;  a_var := Aport; dx_var := DXport; y_var := Yinport; u_var := Uinport;
--    Xinport <= x_var + a_var;
   while (x_var < a_var) loop

      t1 := u_var * dx_var;
      t2 := 3 * x_var;
      t3 := 3 * y_var;
      t4 := t1 * t2;
      t5 := dx_var * t3;
      t6 := u_var - t4;

      y1 := u_var * dx_var;
      u_var := t6 - t5;
      y_var := y_var + y1;
      x_var := x_var + dx_var;
   end loop;

  Xoutport <= x_var;
  Youtport <= y_var;
       t1 := x_var + y_var;
  Uoutport <= t1;
end process P1;

end diffeq;

configuration cfg_DIFFEQ of DIFFEQ is
    for DIFFEQ
    end for;
end cfg_DIFFEQ;
```

APPENDIX (B)

CDFG Files of the Benchmarks

```
*****************************************************
* CDFG for CASCADE generated by 'cdfggen'
*
* Date: Sat May  1 17:02:17 1999
* User: jeonjinh
*****************************************************
(fix 17 15)
edge 0 1 INP REAL 1 32
edge 1 -1 OUTP REAL 1 32
node 0 source - - -
node 1 CASCADE - - -
  length_type INP REAL 32 in - to -
  length_type OUTP REAL 32 out - to -
  subgraph 1
    edge 0 1 INP REAL 1 32
    edge 0 1 INP event 1 1
    edge 1 -1 OUTP REAL 1 32
    node 1 mod_1 - - - (PROCESS)
      length_type STATE0 REAL 32 - - to -
      length_type STATE0_1 REAL 32 - - to -
      length_type STATE0_2 REAL 32 - - to -
      length_type STATE1 REAL 32 - - to -
      length_type STATE1_1 REAL 32 - - to -
      length_type STATE1_2 REAL 32 - - to -
      length_type STATE2 REAL 32 - - to -
      length_type STATE2_1 REAL 32 - - to -
      length_type STATE2_2 REAL 32 - - to -
      length_type STATE3 REAL 32 - - to -
      length_type STATE3_1 REAL 32 - - to -
      length_type STATE3_2 REAL 32 - - to -
      length_type INP1 REAL 32 - - to -
      length_type INP2 REAL 32 - - to -
      length_type INP3 REAL 32 - - to -
      length_type INP4 REAL 32 - - to -
      length_type _tmp_1_5_0_4STATE0 REAL 32 - - to -
      length_type _tmp_0_4STATE0 REAL 32 - - to -
      length_type _tmp_1_6STATE0 REAL 32 - - to -
      length_type _tmp_1_9_0_8INP2 REAL 32 - - to -
      length_type _tmp_0_8INP2 REAL 32 - - to -
      length_type _tmp_1_14_0_13STATE1 REAL 32 - - to -
      length_type _tmp_0_13STATE1 REAL 32 - - to -
      length_type _tmp_1_15STATE1 REAL 32 - - to -
      length_type _tmp_1_18_0_17INP3 REAL 32 - - to -
      length_type _tmp_0_17INP3 REAL 32 - - to -
      length_type _tmp_1_23_0_22STATE2 REAL 32 - - to -
      length_type _tmp_0_22STATE2 REAL 32 - - to -
      length_type _tmp_1_24STATE2 REAL 32 - - to -
      length_type _tmp_1_27_0_26INP4 REAL 32 - - to -
      length_type _tmp_0_26INP4 REAL 32 - - to -
      length_type _tmp_1_32_0_31STATE3 REAL 32 - - to -
      length_type _tmp_0_31STATE3 REAL 32 - - to -
      length_type _tmp_1_33STATE3 REAL 32 - - to -
      length_type _tmp_1_36_0_35OUTP REAL 32 - - to -
      length_type _tmp_0_35OUTP REAL 32 - - to -
      subgraph 1
        edge 0 1 INP REAL 1 32
        edge 0 1 0.005656462366 REAL 1 32
        edge 0 2 0.2855823838 REAL 1 32
        edge 0 2 STATE0_1 REAL 1 32
        edge 1 14 INP1 REAL 1 32
        edge 2 14 _tmp_1_5_0_4STATE0 REAL 1 32
```

```
edge 0 3 -0.9116860618 REAL 1 32
edge 0 3 STATE0_2 REAL 1 32
edge 14 15 _tmp_0_4STATE0 REAL 1 32
edge 3 15 _tmp_1_6STATE0 REAL 1 32
edge 0 4 0.4512591755 REAL 1 32
edge 0 4 STATE0_1 REAL 1 32
edge 15 16 STATE0 REAL 1 32
edge 4 16 _tmp_1_9_0_8INP2 REAL 1 32
edge 16 17 _tmp_0_8INP2 REAL 1 32
edge 0 17 STATE0_2 REAL 1 32
edge 0 30 STATE0_1 REAL 1 32
edge 17 30 depend bool 1 1 (anti)
edge 15 31 STATE0 REAL 1 32
edge 30 31 depend bool 1 1 (anti)
edge 0 5 0.4457342317 REAL 1 32
edge 0 5 STATE1_1 REAL 1 32
edge 17 18 INP2 REAL 1 32
edge 5 18 _tmp_1_14_0_13STATE1 REAL 1 32
edge 0 6 -0.913078673 REAL 1 32
edge 0 6 STATE1_2 REAL 1 32
edge 18 19 _tmp_0_13STATE1 REAL 1 32
edge 6 19 _tmp_1_15STATE1 REAL 1 32
edge 0 7 -1.09869455 REAL 1 32
edge 0 7 STATE1_1 REAL 1 32
edge 19 20 STATE1 REAL 1 32
edge 7 20 _tmp_1_18_0_17INP3 REAL 1 32
edge 20 21 _tmp_0_17INP3 REAL 1 32
edge 0 21 STATE1_2 REAL 1 32
edge 0 32 STATE1_1 REAL 1 32
edge 21 32 depend bool 1 1 (anti)
edge 19 33 STATE1 REAL 1 32
edge 32 33 depend bool 1 1 (anti)
edge 0 8 0.196901665 REAL 1 32
edge 0 8 STATE2_1 REAL 1 32
edge 21 22 INP3 REAL 1 32
edge 8 22 _tmp_1_23_0_22STATE2 REAL 1 32
edge 0 9 -0.9695353354 REAL 1 32
edge 0 9 STATE2_2 REAL 1 32
edge 22 23 _tmp_0_22STATE2 REAL 1 32
edge 9 23 _tmp_1_24STATE2 REAL 1 32
edge 0 10 -0.0099665157 REAL 1 32
edge 0 10 STATE2_1 REAL 1 32
edge 23 24 STATE2 REAL 1 32
edge 10 24 _tmp_1_27_0_26INP4 REAL 1 32
edge 24 25 _tmp_0_26INP4 REAL 1 32
edge 0 25 STATE2_2 REAL 1 32
edge 0 34 STATE2_1 REAL 1 32
edge 25 34 depend bool 1 1 (anti)
edge 23 35 STATE2 REAL 1 32
edge 34 35 depend bool 1 1 (anti)
edge 0 11 0.5515388641 REAL 1 32
edge 0 11 STATE3_1 REAL 1 32
edge 25 26 INP4 REAL 1 32
edge 11 26 _tmp_1_32_0_31STATE3 REAL 1 32
edge 0 12 -0.9705899009 REAL 1 32
edge 0 12 STATE3_2 REAL 1 32
edge 26 27 _tmp_0_31STATE3 REAL 1 32
edge 12 27 _tmp_1_33STATE3 REAL 1 32
edge 0 13 -0.7304169706 REAL 1 32
edge 0 13 STATE3_1 REAL 1 32
edge 27 28 STATE3 REAL 1 32
edge 13 28 _tmp_1_36_0_35OUTP REAL 1 32
edge 28 29 _tmp_0_35OUTP REAL 1 32
edge 0 29 STATE3_2 REAL 1 32
edge 0 36 STATE3_1 REAL 1 32
edge 29 36 depend bool 1 1 (anti)
edge 27 37 STATE3 REAL 1 32
edge 36 37 depend bool 1 1 (anti)
edge 37 -1 STATE3_1 REAL 1 32
edge 36 -1 STATE3_2 REAL 1 32
edge 29 -1 OUTP REAL 1 32
```

```
        edge 35 -1 STATE2_1 REAL 1 32
        edge 34 -1 STATE2_2 REAL 1 32
        edge 33 -1 STATE1_1 REAL 1 32
        edge 32 -1 STATE1_2 REAL 1 32
        edge 31 -1 STATE0_1 REAL 1 32
        edge 30 -1 STATE0_2 REAL 1 32
        node 1 * - - -
        node 2 * - - -
        node 3 * - - -
        node 4 * - - -
        node 5 * - - -
        node 6 * - - -
        node 7 * - - -
        node 8 * - - -
        node 9 * - - -
        node 10 * - - -
        node 11 * - - -
        node 12 * - - -
        node 13 * - - -
        node 14 + - - -
        node 15 + - - -
        node 16 + - - -
        node 17 + - - -
        node 18 + - - -
        node 19 + - - -
        node 20 + - - -
        node 21 + - - -
        node 22 + - - -
        node 23 + - - -
        node 24 + - - -
        node 25 + - - -
        node 26 + - - -
        node 27 + - - -
        node 28 + - - -
        node 29 + - - -
        node 30 = - - -
        node 31 = - - -
        node 32 = - - -
        node 33 = - - -
        node 34 = - - -
        node 35 = - - -
        node 36 = - - -
        node 37 = - - -
        node -1 sink - - -
        node 0 source - - -
      end
    node -1 sink - - -
    node 0 source - - -
  end
node -1 sink - - -
```

```
**************************************************
* CDFG for DIFFEQ generated by 'cdfggen'
*
* Date: Thu Dec 23 16:53:59 1999
* User: jeonjinh
**************************************************
edge 0 1 XINPORT INTEGER 1 32
edge 0 1 DXPORT INTEGER 1 32
edge 0 1 APORT INTEGER 1 32
edge 0 1 YINPORT INTEGER 1 32
edge 0 1 UINPORT INTEGER 1 32
edge 1 -1 XOUTPORT INTEGER 1 32
edge 1 -1 YOUTPORT INTEGER 1 32
edge 1 -1 UOUTPORT INTEGER 1 32
node 0 source - - -
node 1 DIFFEQ - - -
```

```
length_type XINPORT INTEGER 32 in - to -
length_type XOUTPORT INTEGER 32 out - to -
length_type DXPORT INTEGER 32 in - to -
length_type APORT INTEGER 32 in - to -
length_type YINPORT INTEGER 32 in - to -
length_type YOUTPORT INTEGER 32 out - to -
length_type UINPORT INTEGER 32 in - to -
length_type UOUTPORT INTEGER 32 out - to -
subgraph 1
  edge 0 1 XINPORT INTEGER 1 32
  edge 0 1 APORT INTEGER 1 32
  edge 0 1 DXPORT INTEGER 1 32
  edge 0 1 YINPORT INTEGER 1 32
  edge 0 1 UINPORT INTEGER 1 32
  edge 0 1 APORT event 1 1
  edge 0 1 DXPORT event 1 1
  edge 0 1 XINPORT event 1 1
  edge 0 1 YINPORT event 1 1
  edge 0 1 UINPORT event 1 1
  edge 1 -1 UOUTPORT INTEGER 1 32
  edge 1 -1 YOUTPORT INTEGER 1 32
  edge 1 -1 XOUTPORT INTEGER 1 32
  node 1 P1 - - - (PROCESS)
    length_type X_VAR INTEGER 32 - - to -
    length_type Y_VAR INTEGER 32 - - to -
    length_type U_VAR INTEGER 32 - - to -
    length_type A_VAR INTEGER 32 - - to -
    length_type DX_VAR INTEGER 32 - - to -
    length_type X1 INTEGER 32 - - to -
    length_type Y1 INTEGER 32 - - to -
    length_type T1 INTEGER 32 - - to -
    length_type T2 INTEGER 32 - - to -
    length_type T3 INTEGER 32 - - to -
    length_type T4 INTEGER 32 - - to -
    length_type T5 INTEGER 32 - - to -
    length_type T6 INTEGER 32 - - to -
    subgraph 1
      edge 0 2 XINPORT INTEGER 1 32
      edge 0 3 APORT INTEGER 1 32
      edge 0 4 DXPORT INTEGER 1 32
      edge 0 5 YINPORT INTEGER 1 32
      edge 0 6 UINPORT INTEGER 1 32
      edge 2 10 X_VAR INTEGER 1 32
      edge 3 10 A_VAR INTEGER 1 32
      edge 6 10 U_VAR INTEGER 1 32
      edge 4 10 DX_VAR INTEGER 1 32
      edge 5 10 Y_VAR INTEGER 1 32
      edge 10 7 X_VAR INTEGER 1 32
      edge 10 8 Y_VAR INTEGER 1 32
      edge 10 1 X_VAR INTEGER 1 32
      edge 10 1 Y_VAR INTEGER 1 32
      edge 1 9 T1 INTEGER 1 32
      edge 9 -1 UOUTPORT INTEGER 1 32
      edge 8 -1 YOUTPORT INTEGER 1 32
      edge 7 -1 XOUTPORT INTEGER 1 32
      node 1 + 1 - -
      node 2 = - - -
      node 3 = - - -
      node 4 = - - -
      node 5 = - - -
      node 6 = - - -
      node 7 = - - -
      node 8 = - - -
      node 9 = - - -
      node 10 loop - - -
        iteration 6
          edge 0 1 X_VAR INTEGER 1 32
          edge 0 1 A_VAR INTEGER 1 32
          edge 1 -1 ctrl bool 1 1
          edge 0 1 endloop ctrl 1 1
          node 1 < 1 - -
```

```
            node -1 sink - - -
            node 0 source - - -
          end
          subgraph 6
            edge 0 1 U_VAR INTEGER 1 32
            edge 0 1 DX_VAR INTEGER 1 32
            edge 0 2 3 INTEGER 1 32
            edge 0 2 X_VAR INTEGER 1 32
            edge 0 3 3 INTEGER 1 32
            edge 0 3 Y_VAR INTEGER 1 32
            edge 1 4 T1 INTEGER 1 32
            edge 2 4 T2 INTEGER 1 32
            edge 0 5 DX_VAR INTEGER 1 32
            edge 3 5 T3 INTEGER 1 32
            edge 0 9 U_VAR INTEGER 1 32
            edge 4 9 T4 INTEGER 1 32
            edge 0 6 U_VAR INTEGER 1 32
            edge 0 6 DX_VAR INTEGER 1 32
            edge 9 10 T6 INTEGER 1 32
            edge 5 10 T5 INTEGER 1 32
            edge 6 10 depend bool 1 1 (anti)
            edge 0 7 Y_VAR INTEGER 1 32
            edge 6 7 Y1 INTEGER 1 32
            edge 3 7 depend bool 1 1 (anti)
            edge 0 8 X_VAR INTEGER 1 32
            edge 0 8 DX_VAR INTEGER 1 32
            edge 2 8 depend bool 1 1 (anti)
            edge 8 -1 X_VAR INTEGER 1 32
            edge 10 -1 U_VAR INTEGER 1 32
            edge 7 -1 Y_VAR INTEGER 1 32
            edge 8 -1 endloop ctrl 1 1
            node 1 * 2 - -
            node 2 * 2 - -
            node 3 * 2 - -
            node 4 * 2 - -
            node 5 * 2 - -
            node 6 * 2 - -
            node 7 + 1 - -
            node 8 + 1 - -
            node 9 - 1 - -
            node 10 - 1 - -
            node -1 sink - - -
            node 0 source - - -
          end
        node -1 sink - - -
        node 0 source - - -
      end
    node -1 sink - - -
    node 0 source - - -
  end
node -1 sink - - -
```

APPENDIX (C)

Source Code of Activity-Sensitive Clock Design

//Due to space limit, only the header files and main function are provided here for reference. For the //implementation, please contact the author at kang_changjun@hotmail.com

//There are two application packages in Activity-Sensitive Clock Design. The first one *tcdfg* creates the activity information of each module. By reading the output of *tcdfg*, the second one *buildclk* constructs the clock tree.

//Make File

```
CC = g++
CXXFLAGS = -g  -fno-inline -fno-default-inline
CLASSOBJ = hierarchyNode.o  conIteNode.o operationNode.o sourceSink.o \
        trueFalCase.o activity.o
OBJ = tcdfg.o cdfg.o  cdfgNode.o edge.o $(CLASSOBJ)

tcdfg: $(OBJ)
        $(CC) -o tcdfg $(CXXFLAGS) $(OBJ)
tcdfg.o: cdfg.h
cdfgNode.o: cdfgNode.h
edge.o: edge.h
cdfg.o: cdfg.h
hierarchyNode.o: hierarchyNode.h
conIteNode.o: conIteNode.h
operationNode.o: operationNode.h
sourceSink.o: sourceSink.h
trueFalCase.o: trueFalCase.h
activity.o: activity.h

clean :

        rm $(OBJ)
        rm tcdfg
```

```
#ifndef ACTIVITY_H
#define ACTIVITY_H
#include <vector>
#include"enumType.h"
#include <fstream.h>
//activity describe activity pattern
class activity
{
 public:
  activity(int=0, operationType=o_others, int=0);
  operationType getOpType() {return opType;}
  void setActPattern(int i) {actPattern[i]=1;}
  int getNumber() {return number;}
  void printActPattern(ofstream &output);
```

```
 private:
  int number;
  operationType opType;
  vector<int> actPattern;
};

#endif




#ifndef CDFG_H
#define CDFG_H

#include "cdfgNode.h"
#include "edge.h"
#include <stack.h>
#include "operationNode.h"
#include "hierarchyNode.h"
#include "conIteNode.h"
#include "trueFalCase.h"
#include "sourceSink.h"
#include "activity.h"
#include <fstream.h>
#include <iterator.h>
#include <iostream.h>
#include <fstream.h>
#include <assert.h>
#include  <typeinfo>
#define lineSize 200
#include <algorithm>
#include <cstring>
#include <iomanip>
typedef std::multimap<operationType,int, std::less<int> > leastReqMap;
//class cdfg is the data structure for CDFG
class cdfg
{
 public:
  cdfg() { root=0;}
  void findHelp( cdfgNode * &,int , const vector < int > & , cdfgNode * &);
  cdfgNode * find(int , const  vector < int > &);
  void findParentHelp(cdfgNode *,int , const  vector < int > & , cdfgNode * &parent);
  cdfgNode * findParent(int ,  const vector < int > &);

  cdfgNode * psFirstNode(int,  vector< int > &,  stack< cdfgNode *> &,  edge *);
  cdfgNode * psSecondNode(int, int, vector< int > &,  stack< cdfgNode *> &,  edge *&);

  void psEdgeLine( ifstream &, vector< int > &, stack < cdfgNode *> &);
  void psOperationNode(int id, generalType genType,vector<int> &hie);
  operationNode *replaceByOpNode(cdfgNode * original);
  void splitHieNode(cdfgNode * original, vector< int > &hie,stack < cdfgNode *> &begin,
  stack<cdfgNode *> &end);

  void psNodeLine(int id, string name, vector< int > &hie, stack < cdfgNode *> &begin, stack<cdfgNode
  *>&end);
  void psHieNode(int id, generalType genType,vector<int> &hie,stack < cdfgNode *> &begin,
  stack<cdfgNode *> &end);
  void psConIteNode(int id, generalType genType,vector<int> &hie,stack < cdfgNode *> &begin, )
  stack<cdfgNode *> &end);
  void psConIteHelp(cdfgNode *conIte, vector<int> &hie,stack < cdfgNode *> &begin);
  void psTrueFalCase(int id, generalType genType, vector<int> &hie,stack < cdfgNode *> &begin,
  stack<cdfgNode *> &end);
  void psTrueFalCaseHelp(vector<int> &hie,stack< cdfgNode *> &begin, stack<cdfgNode *> &end);
  sourceSink *replaceBySouSink(cdfgNode * original);
  void psSourceSink(int id, generalType genType,vector<int> &hie);
  void psEnd ( ifstream &input, long begOfLine, vector<int> &hie,stack < cdfgNode *> &begin,
  stack<cdfgNode *> &end);
  int prePsNodeLine(ifstream &input, const string &firstWord ,string &name, long begOfLine,stack<
  cdfgNode *> &begin);
```

```
  void psLine(ifstream &input, vector< int > &hie,stack < cdfgNode *> &begin, stack<cdfgNode *>
  &end);
  //bool isCaseLoopBody(const string &firstWord,  stack < cdfgNode *> &begin );
  void psFile();
  void printAllNode(cdfgNode *top, ofstream &outPut);
  generalType convertString( const string & i);
  cdfgNode * getRoot() { return root; }
  cdfgNode * setRoot(cdfgNode *i) { return root=i; }
  bool curIsFalseBody(stack < cdfgNode *> &begin);
  bool curIsLoopBody(stack < cdfgNode *> &begin);
  bool curIsTrueBody(stack < cdfgNode *> &begin);
  bool curIsCaseBody(stack < cdfgNode *> &begin);
  bool nextIsCaseBody(const string &firstWord,  stack < cdfgNode *> &begin );
  bool nextIsFalseBody(const string &firstWord,  stack < cdfgNode *> &begin );
  bool isLoopBody(const string &firstWord,  stack < cdfgNode *> &begin );
  bool isCase(const string &firstWord,  stack < cdfgNode *> &begin );
  int prePsEnd(ifstream &input, long begOfLine,stack < cdfgNode *> &begin);
  void popOut( vector<int> &hie,stack < cdfgNode *> &begin, stack<cdfgNode *> &end);
  void addPsuedEdge(cdfgNode *front, cdfgNode *back, vector<int> &hie );
  void splitTFCNode(cdfgNode * original, vector< int > &hie,stack < cdfgNode *> &begin,
  stack<cdfgNode *> &end);
  void splitConIteNode(cdfgNode * original, vector< int > &hie,stack < cdfgNode *> &begin,
  stack<cdfgNode *> &end);
  void disconnectTFC(hierarchyNode *cond, cdfgNode *branchDel);
  void connectTFC(hierarchyNode *cond, cdfgNode *branch);
  void selectTFC(hierarchyNode *cond);
  void conditionAhead(hierarchyNode *cond);
  void prePsCond(cdfgNode *top);
  void ASAPHelp(cdfgNode *top);
  void allScheduled(cdfgNode *top, bool &flag);
  void ASAP();
  void sortOfAsap(cdfgNode *top,mMap &sch);
  void printAsap(mMap &sch);
  void leastReqMod(mMap &sch, leastReqMap &leastReq);
  void outputActPatt(mMap &sch, leastReqMap &leastReq);
  private:
  cdfgNode *root;
};

#endif
```

```
#ifndef CDFG_NODE_H
#define CDFG_NODE_H

#include <vector>
#include "edge.h"
#include <map>
#include "enumType.h"
typedef std::multimap< int,operationType, std::less<int> > mMap;
class cdfgNode;
//class cdfgNode describes nodes in CDFG
class cdfgNode
{
  //friend class cdfg;
 protected:

  int id;
  vector< int > hierarchy;
  vector < cdfgNode * >  nextNodePtr;
  vector < edge * >  edgePred;
  vector < edge *>  edgeSucc;
  vector < int > ctrStep;
  bool scheduled;

 public:
  cdfgNode();
  cdfgNode( int, const vector<int> &, const vector<cdfgNode *> &,
```

```
          const  vector<edge *> &, const vector<edge *> &, const vector<int> &, bool );
  cdfgNode(const cdfgNode &);
  virtual ~cdfgNode() {}
  bool operator ==(const cdfgNode &);
  bool sameNode( int i, const vector< int > &hie);
  //int setId(int i) { return id=i;}
  int setId(int i);
  //int getId() { return id;}
  int getId();
  // vector< int > setHierarchy(const vector <int> &hie) {return hierarchy=hie;}
  vector< int > setHierarchy(const vector <int> &hie);
  //vector< int > getHierarchy()  { return hierarchy;}
  vector< int > &getHierarchy();
  //vector<cdfgNode *> getNextNode() { return nextNodePtr;}
  vector<cdfgNode *> &getNextNode();
  //vector<edge *> getPredEdge() { return edgePred;}
  vector<edge *> &getPredEdge();
  //vector<edge *> getSuccEdge() { return edgeSucc;}
  vector<edge *> &getSuccEdge();
  bool getScheduled() {return scheduled;}
  bool setScheduled(bool x) {return scheduled=x;}
  virtual string getNodeType() {}
  int  scheduleHelp();
  virtual void schedule() {}
  virtual cdfgNode *isCondNode() {return 0;}
  virtual cdfgNode *isTrueNode() {return 0;}
  virtual cdfgNode *isFalseNode() {return 0;}
  virtual cdfgNode *isCaseNode() {return 0;}
  virtual void makeMultiMap(mMap &sch) {}
  void addPredEdge(edge *edgePtr);
  void addSuccEdge(edge *edgePtr);
  void addNextNodePtr(cdfgNode *nodePtr);
};

#endif



#ifndef EDGE_H
#define EDGE_H

#include <vector.h>
#include <string>

class cdfgNode;
class edge;
//class edge describe edges in CDFG
class edge
{
  // friend class cdfg;
 private:
  string name;
  vector<int> hierarchy;
  vector <cdfgNode *> nodePred;
  vector <cdfgNode *> nodeSucc;
  static int pusedoCount;

 public:
  edge():name(),hierarchy(),nodePred(),nodeSucc() {}
  edge *psuedo(char *,  vector<int> &, cdfgNode *&, cdfgNode *&);
  ~edge() {}
  string setName( const string &right) { return name=right;}
  string getName() { return name;}
  vector< int > setHierarchy(const vector <int> &hie) {return hierarchy=hie;}
  vector< int > &getHierarchy()  { return hierarchy;}
  vector< cdfgNode *> &getPredNode() { return nodePred;}
  vector< cdfgNode *> &getSuccNode() { return nodeSucc;}
  void addPredNode( cdfgNode *nodePtr);
  void addSuccNode( cdfgNode *nodePtr);
```

```
};

#endif



ifndef ENUMTYPE_H
#define ENUMTYPE_H

enum hierarchyType {
   module, loop, cond, h_others
};

enum operationType
{
  adder, subs, multiplier, divider, multiplexer, comparator, o_others
};

enum conIteType
{
  condition, iteration, c_others
};

enum trueFalCaseType
{
  trueType, falseType, caseType, loopBodyType, t_others
};

enum sourceSinkType
{
  source, sink, s_others
};

enum begEndType
{
  b_begin, b_end, b_others
};


enum generalType
{
     g_eq ,
     g_neq,
     g_lt ,
     g_ltEq ,
     g_gt,
     g_gtEq ,
     g_add,
     g_sub,
     g_multiply ,
     g_div,
     g_modules,
     g_abs ,
     g_assign ,
     g_bAnd,
     g_bOr,
     g_nand ,
     g_nor ,
     g_xor ,
     g_concat ,
     g_exp ,
     g_not,

     g_loop,
     g_cond,
     g_module,

     g_condition,
     g_iteration,
```

```
    g_case,
    g_true,
    g_false,
    g_loopBody,

    g_source,
    g_sink,

    g_edge,
    g_node,

    g_subgraph,
    g_end,

    g_others
};
#endif


#ifndef HIERARCHY_NODE
#define HIERARCHY_NODE

#include "cdfgNode.h"
#include "enumType.h"
#include "trueFalCase.h"
#include "conIteNode.h"
#include "edge.h"
#include "algorithm"

class hierarchyNode;
//class hierarchyNode describe module, condition and loop
class hierarchyNode: public cdfgNode
{
 public:
  hierarchyNode(): cdfgNode()  { begEnd=b_others; typeName=h_others;}
  hierarchyNode(const cdfgNode &right): cdfgNode(right) { begEnd=b_others; typeName=h_others;}
  begEndType setBegEnd( begEndType i) { return begEnd=i;}
  begEndType getBegEnd() { return begEnd;}
  hierarchyType setHierarchyType( hierarchyType type) { return typeName=type;}
  hierarchyType getHierarchyType() { return typeName;}
  virtual string getNodeType() {string type("hierarchyNode"); return type;}
  virtual ~hierarchyNode() {}
  virtual void schedule();

  virtual hierarchyNode *isCondNode();
  cdfgNode *isThereTrue();
  cdfgNode *isThereFalse();
  cdfgNode *isThereCondition();
  int howManyCases();
  cdfgNode *selectedCase(int);
  void delTrueFalCase(cdfgNode *entrance);
  virtual void makeMultiMap(mMap &sch) {}
 private:
  begEndType begEnd;
  hierarchyType typeName;          //module, loop or cond
};
#endif




#ifndef OPERATION_NODE
#define OPERATION_NODE

#include "cdfgNode.h"
#include "enumType.h"
//class operationNode describes operation node in CDFG
class operationNode : public cdfgNode
```

```
{
 public:
  operationNode(): cdfgNode()  { typeName=o_others;}
  operationNode(const cdfgNode &right):cdfgNode(right) { typeName=o_others;}
  operationType setOperationType(operationType i) {return typeName=i;}
  operationType getOperationType() {return typeName;}
  virtual string getNodeType() {string type("operationNode"); return type;}
  virtual ~operationNode() {}
  virtual void schedule();
  virtual void makeMultiMap(mMap &sch);

 private:
  operationType typeName;          // adder, multiplier, comparator etc
};
#endif
```

```
#ifndef SOURCESINK_H
#define SOURCESINK_H

#include "cdfgNode.h"
#include "enumType.h"
//class sourceSink desribes starting node and ending node
class sourceSink: public cdfgNode
{
 public:
  sourceSink():cdfgNode() {typeName=s_others;}
  sourceSink(const cdfgNode &right,:cdfgNode(right) {typeName=s_others;}
  sourceSinkType setSourceSinkType ( sourceSinkType i) { return typeName=i;}
  sourceSinkType getSourceSinkType () { return typeName;}
  virtual string getNodeType() {string type("sourceSink"); return type;}
  virtual ~sourceSink() {}
  virtual void schedule();
  virtual void makeMultiMap(mMap &sch) {}
 private:
  sourceSinkType typeName;
};
#endif
```

```
#ifndef TRUEFALCASE_H
#define TRUEFALCASE_H

#include "cdfgNode.h"
#include "enumType.h"
//class trueFalCase desribes condition
class trueFalCase: public cdfgNode
{
 public:
  trueFalCase():cdfgNode()  { typeName=t_others;selected=false;}
  trueFalCase(const cdfgNode &right ):cdfgNode(right)  { typeName=t_others;selected=false;}
  trueFalCaseType setTrueFalCaseType( trueFalCaseType i) { return typeName=i;}
  trueFalCaseType getTrueFalCaseType( ) { return typeName;}
  bool setSelect(bool i) { selected=i;}
  bool getSelect() { return selected; }
  virtual string getNodeType() {string type("trueFalCase"); return type;}
  virtual ~trueFalCase() {}
  virtual void schedule();

  virtual cdfgNode *isTrueNode();
  virtual cdfgNode *isFalseNode();
  virtual cdfgNode *isCaseNode();
  virtual void makeMultiMap(mMap &sch) {}
 private:
```

```
  trueFalCaseType typeName;
  bool selected;            //for scheduling, either true or false branch is selected
};
#endif
```

```
#include "cdfg.h"
//#include "fstream.h"

int main ()
{
  cdfg x;
  x.psFile();
  ofstream outTest("testFile.dat", ios::out);
  if( !outTest )
    {
       cerr << "File could not be opened" << endl;
       exit(1);
    }
   cdfgNode * root= x.getRoot();
   x.printAllNode(root, outTest);
   x.prePsCond(root);
   x.ASAP();
    mMap sch;
   x.sortOfAsap(root,sch);
   x.printAsap(sch);
   leastReqMap leastReq;
   x.leastReqMod(sch,leastReq);
   for( leastReqMap::const_iterator it=leastReq.begin(); it!=leastReq.end(); it++)
     cout<<it->first<<"    "<<it->second<<'\n';
   x.outputActPatt(sch,leastReq);
  return 0;
}
```

//Make File

```
CC = g++
CXXFLAGS = -g  -fno-inline -fno-default-inline
CLASSOBJ = node.o level.o pairnode.o pairnodelist.o  clktree.c
OBJ =  merge.o  $(CLASSOBJ)

buildclk: $(OBJ)
       $(CC) -o buildclk $(CXXFLAGS) $(OBJ)


node.o: node.h
level.o: level.h
pairnode.o: pairnode.h
parinodelist.o: pairnodelist.h
clktree.o: clktree.h
merge.o: merge.cc clktree.h enumType.h

clean :

       rm $(OBJ)
       rm buildclk
```

```
#ifndef CLKTREE_H
#define CLKTREE_H
#include "level.h"
typedef vector<level>::iterator lItr;
typedef vector<level>::reverse_iterator rlItr;
```

```
class clkTree: public vector<level>
{
 public:
  clkTree() {}
  void buildTree(level bottom);
  void buildTreeRandomly(level bottom);
  double pwrInterconTree();
  double pwrOfLeaves();
  double totPwr();
  int lenCtrTree();
  void optimize();
  void outputFile(ofstream &output);
};
#endif
```

```
#ifndef ENUMTYPE_H
#define ENUMTYPE_H

enum hierarchyType {
   module, loop, cond, h_others
};

enum operationType
{
  adder, subs, multiplier, divider, multiplexer, comparator, o_others
};

enum conIteType
{
  condition, iteration, c_others
};

enum trueFalCaseType
{
  trueType, falseType, caseType, loopBodyType, t_others
};

enum sourceSinkType
{
  source, sink, s_others
};

enum begEndType
{
  b_begin, b_end, b_others
};
```

```
#ifndef LEVEL_H
#define LEVEL_H
#include "node.h"
#include <time.h>
#include <fstream>
typedef vector<node>::iterator nodeItr;
class level;

class level: public vector<node>
{
public:
  level();
  level(const level &right);
  //  level &operator = (const level &right);
  void determineCtr(level &parentLev);
  level  parentLevel();
  void outputFile(ofstream &output);
```

```
  int transitCtrLev();
  int logDisLevel();
  int curLevel();
  int totLevel();
  double weightOfClk(int totLev); //current level clk weight
  double weightOfClk(int lev, int totLev); //level #lev clk weight
  double weightOfCtr(int totLev);
  double weightOfCtr(int lev, int totLev);
  int wgtLength(int totLev);
  void creatBottomLevel();
  double pwrInterconLevel(int totLevel);
  double pwrOfLeaves();
  int lenCtrLevel(int totLevel);
  nodeItr findByMark(int m);
};

#endif



#ifndef NODE_H
#define NODE_include
#include <vector>
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <algorithm>
#include <math.h>
#include <stdlib.h>

#define NoneEx -1
#define unitPwr 0.4   //power consumed over a leaf clock wire(unit wirelength)

#define pwrAdder 4
#define pwrSubs 4
#define pwrMultiplier 8
#define pwrDivider 8
#define pwrMultiplexer 3
#define pwrComparator 4
#define cycles 4

class node;

typedef vector<bool>::iterator bItr;
typedef vector<bool>::difference_type difference;

class node
{
  friend ostream &operator<<(ostream &output, const node &node1);
 public:
  node();
  node(int length);
  node(const node &node1);
  node(vector<bool> act, double pwrMod, int i);
  node(int length, double actDensity, double pwr=pwrAdder, int m=NoneEx);
  vector<bool> getActPattern();
  vector<bool> setActPattern( const vector<bool> &pattern);
  vector<bool> getControl();
  vector<bool> setControl(const vector<bool> &ctr);
  int getMark();
  int setMark(int);
  int getParentMark();
  int setParentMark(int);
  int getLeftMark();
  int setLeftMark(int);
  int getRightMark();
  int setRightMark(int);
  double getPowerOfLeaf();
  double setPowerOfLeaf(double pwr);
  node getParent(node &node1);
```

```
  int logicDistance(const node &node1);
  double pwrMerging(node &node1,double wgtOfClk, double wgtOfCtr);
  node &operator =(const node &right);
  bool determineCtrHead(const node &parent);
  void determineCtr(const node &parent);
  int transitCtr();
  int transitClk();
  void childAct(const node &parent);
  void setCtr(bItr start, bItr end);
  bool controlAlwaysHigh();
  bool controlAlwaysLow();
  double pwrOfNode(double wgtOfClk, double wgtOfCtr);
  void changeAssociate(const node &parent, node &leftChd, node &rightChd);
  void optimizeNode(const node &parent, node &leftChd, node & rightChd,
                    double wgtOfClk, double wgtOfCtr, double wgtChdClk, double wgtChdCtr);
  void evalBottomNode(difference start, difference end,const node &parent, double wgtOfClk, double
wgtOfCtr);

  void evaluate(difference start, difference end, const node &parent, node &leftChd, node &rightChd,
                    double wgtOfClk, double wgtOfCtr, double wgtChdClk, double wgtChdCtr);
  void optBottomNode(const node &parent,double wgtOfClk, double wgtOfCtr);

 private:
  vector<bool> actPattern;
  vector<bool> control;
  int mark;
  int leftMark;
  int rightMark;
  int parentMark;
  double powerOfLeaf;
};
#endif



#ifndef PAIRNODE_H
#define PAIRNODE_H

#include <iostream.h>
#include <iomanip.h>

class pairNode
{
        friend ostream &operator<<(ostream &output, const pairNode &node1);
public:
        pairNode(int seq1=0, int seq2=0, double pwrMg=0);
        pairNode(const pairNode &right);
        double getPwrMg();
        int getMember1();
        int getMember2();
        bool operator<(const pairNode &comp);
        pairNode &operator=(const pairNode &right);
        bool overlap(pairNode &p1);
private:
        double pwrMerge;
        int nodeMember1;
        int nodeMember2;
};
#endif



#ifndef PAIRNODELIST_H
#define PAIRNODELISH_H

#include "pairnode.h"
#include "level.h"
#include <list>
typedef list<pairNode>::iterator pItr;
```

```
class pairNodeList: public list<pairNode>
{
public:
        pairNodeList();
        void buildSortedList(level &lev,int totLevel);
        void getPairList();
        int unPairNode(const level &lev);
        void appendUnpair(int unpair);
        level pairedLevel(level &lev, int totLevel);
        void outputFile(ofstream &output);

};
#endif
```

```
#include "enumType.h"
#include "clktree.h"
#include <string>
int randCase();
int benchCase();
vector<bool> convertAct(string s);
level readFromBench(ifstream &input);

int main()
{
  int select;
  cout<<"choose the activity pattern source"<<endl;
  cout<<"select 0 for randomly generating\n"
      <<"select 1 for benchmark"<<endl;
  cout<<"select:";
  cin>>select;
  if(select==0)
    randCase();
  else
    benchCase();
}


int randCase()
{
  double pwrAfterMerge, pwrAfterOpt, pwrByRandom;
  double pwrLeafMerge, pwrLeafOpt, pwrLeafRandom;
  int lenCtrAfterMerge,  lenCtrAfterOpt,lenCtrRandom;

  level bottom;
  clkTree mergeTree, randTree;
  bottom.creatBottomLevel();
  mergeTree.buildTree(bottom);
  pwrAfterMerge=mergeTree.totPwr();
  pwrLeafMerge=mergeTree.pwrOfLeaves();
  lenCtrAfterMerge=mergeTree.lenCtrTree();
  ofstream outPut("tree.data", ios::out);
  if (!outPut)
    {
      cerr<<"could not open the file"<<endl;
      exit(1);
    }
  mergeTree.outputFile(outPut);

  mergeTree.optimize();
  pwrAfterOpt=mergeTree.totPwr();
  pwrLeafOpt=mergeTree.pwrOfLeaves();
  lenCtrAfterOpt=mergeTree.lenCtrTree();
  mergeTree.outputFile(outPut);

  randTree.buildTreeRandomly(bottom);
  pwrByRandom=randTree.totPwr();
  pwrLeafRandom=randTree.pwrOfLeaves();
```

```
  lenCtrRandom=randTree.lenCtrTree();
  randTree.outputFile(outPut);

  cout<<setw(30)<<setiosflags(ios::right)<<"After merging"
      <<setw(15)<<"after Opt"<<setw(15)<<"by random"<<endl;
  cout<<setw(15)<<"total power"<<setw(15)<<pwrAfterMerge
      <<setw(15)<<pwrAfterOpt<<setw(15)<<pwrByRandom<<endl;
  cout<<setw(15)<<"leaf power"<<setw(15)<<pwrLeafMerge
      <<setw(15)<<pwrLeafOpt<<setw(15)<<pwrLeafRandom<<endl;
  cout<<setw(15)<<"control length"<<setw(15)<<lenCtrAfterMerge
      <<setw(15)<<lenCtrAfterOpt<<setw(15)<<lenCtrRandom<<endl;
  return 0;
}

int benchCase()
{
  double pwrAfterMerge, pwrAfterOpt, pwrByRandom;
  double pwrLeafMerge, pwrLeafOpt, pwrLeafRandom;
  int lenCtrAfterMerge,  lenCtrAfterOpt,lenCtrRandom;

  level bottom;
  clkTree mergeTree, randTree;
  ifstream inputFile("activityPattern.dat", ios::in);
  if(!inputFile)
    {
      cerr<<"file could not be opened"<<endl;
      exit(1);
    }
  bottom=readFromBench(inputFile);

  mergeTree.buildTree(bottom);
  pwrAfterMerge=mergeTree.totPwr();
  pwrLeafMerge=mergeTree.pwrOfLeaves();
  lenCtrAfterMerge=mergeTree.lenCtrTree();
  ofstream outPut("benchOut.data", ios::out);
  if (!outPut)
    {
      cerr<<"could not open the file"<<endl;
      exit(1);
    }
  mergeTree.outputFile(outPut);

  mergeTree.optimize();
  pwrAfterOpt=mergeTree.totPwr();
  pwrLeafOpt=mergeTree.pwrOfLeaves();
  lenCtrAfterOpt=mergeTree.lenCtrTree();
  mergeTree.outputFile(outPut);

  randTree.buildTreeRandomly(bottom);
  pwrByRandom=randTree.totPwr();
  pwrLeafRandom=randTree.pwrOfLeaves();
  lenCtrRandom=randTree.lenCtrTree();
  randTree.outputFile(outPut);

  cout<<setw(30)<<setiosflags(ios::right)<<"After merging"
      <<setw(15)<<"after Opt"<<setw(15)<<"by random"<<endl;
  cout<<setw(15)<<"total power"<<setw(15)<<pwrAfterMerge
      <<setw(15)<<pwrAfterOpt<<setw(15)<<pwrByRandom<<endl;
  cout<<setw(15)<<"leaf power"<<setw(15)<<pwrLeafMerge
      <<setw(15)<<pwrLeafOpt<<setw(15)<<pwrLeafRandom<<endl;
  cout<<setw(15)<<"control length"<<setw(15)<<lenCtrAfterMerge
      <<setw(15)<<lenCtrAfterOpt<<setw(15)<<lenCtrRandom<<endl;

  cout<<"power saving over clock: "<<(pwrByRandom-pwrLeafRandom-pwrAfterOpt+pwrLeafOpt)/(pwrByRandom-
pwrLeafRandom)*100<<endl;
  cout<<"power increase over modules: "<<(pwrLeafRandom-pwrLeafOpt)/pwrLeafRandom*100<<endl;
  cout<<"ctrl length saving: "<<( (double) (lenCtrRandom-lenCtrAfterOpt))/lenCtrRandom*100<<endl;
  return 0;
}

vector<bool> convertAct(string s)
```

```
{
  vector<bool> actPattern;
  for(int i=0; i<s.size(); i++)
    {
      if( s[i]=='1' )
        actPattern.push_back(true);
      else if ( s[i]=='0' )
        actPattern.push_back(false);
      else
        cerr<<"wrong bit in activity pattern";
    }
  return actPattern;
}

level readFromBench(ifstream &input)
{
  int num, type;
  string act;
  level bottom;
  int mark=0;
  vector<bool> actPattern;
  node *nodePtr;
  while(!input.eof())
    {
      input>>num>>type>>act;
      actPattern=convertAct(act);
      switch((operationType)type)
        {
        case adder :  nodePtr = new node(actPattern, pwrAdder, mark++);  break;
        case subs :   nodePtr = new node(actPattern, pwrSubs, mark++);  break;
        case multiplier : nodePtr = new node(actPattern, pwrMultiplier, mark++);  break;
        case divider :  nodePtr = new node(actPattern, pwrDivider, mark++);  break;
        case multiplexer : nodePtr = new node(actPattern, pwrMultiplexer, mark++);  break;
        case comparator : nodePtr = new node(actPattern, pwrComparator, mark++);  break;
        default:   cerr<<"no such operation type";
        }
      bottom.push_back(*nodePtr);
    }
  return bottom;
}
```

APPENDIX (D)

Source Code of Clock Frequency Reduction

//Make File

```
CXX = g++

CXXFLAGS = -g  -fno-inline -fno-default-inline

INCLUDE = -I../CDFG/parser/include/
OBJ = extInfo.o asapsch.o bind.o  mainf.o component.o assign.o \
      resource.o energy.o chain.o multiclk.o
LDDIR = -L../CDFG/parser/lib/
LIB = -lcdfg
INCLUDEDIR = ../CDFG/parser/include
.cc.o:
       $(CXX) -c $(CXXFLAGS) $(INCLUDE) $<
schedule: $(OBJ)
       $(CXX) -o schedule $(CXXFLAGS)  $(INCLUDE)  $(LDDIR)  $(OBJ) $(LIB)
clean :

       rm $(OBJ)
       rm schedule

extInfo.o:extInfo.h
bind.o: bind.h
asapsch.o: asapsch.h
mainf.o:asapsch.h critpath.h
component.o:component.h
resource.o:resource.h
assign.o:assign.h
energy.o:energy.h
chain.o:chain.h
multiclk.o:multiclk.h

-----------------------------------------------------------------------------------------------

//ASAP sheduling
#ifndef ASAPSCH_H
#define ASAPSCH_H

#include "extInfo.h"
#include <math.h>
#include <iostream.h>
#include <fstream>
using std::ifstream;
#include <vector>
#include <map>
typedef  std::map< int , int, std::less<int>  >   opSource;
//first int is opkind, second int is quantity
typedef  std::map< int, opSource , std::less<int>  >  srcByCycle;
//int refers to the clock cycle
#include "bind.h"

opSource hardwareSrs(ifstream &input);
void UpdateList(NodePtrList &list, Node *n, int mark);
void doubleSource(srcByCycle &hardware, const opSource &available);
int determineTs( srcByCycle &hardware, const opSource &available, int clock,
                 int tend, toplib &mylib, Node *cur );
void ASAP(Subgraph *subg, int ts, srcByCycle &hardware, const opSource &available,
          toplib &mylib, int clock, Subgraph *root);
void printAllNodes(Subgraph *subg);
NodePtrList getVtualPreds(Subgraph *subg, Node *cur,NodePtrList &vtualPreds);
#endif
```

```
//Resource allocation
#ifndef ASSIGN_H
#define ASSIGN_H

#include "component.h"
#include "resource.h"
#include <time.h>

bool isOverlap(Node *cur, vector<Node *> &vect, int bind, toplib &mylib, int clock);
bool isInVector(Node *cur, vector<Node *> vect);
vector<leaf> distrSameOpKind( vector<preAllocLeaf> &sameOpKind, toplib &mylib, int clock );
vector<leaf> assignOpKind( vector<preAllocLeaf> &modules, toplib &mylib, int clock, opSource
hardwareSrs );
void distSameOpHelp(Node *cur, vector<Node *> &vect, vector<leaf> &potentialLeaves,toplib &mylib, int
clock );
vector<Node *> getOverlapNodes( Node *cur, vector<Node *> &vect);
vector<int> closerBind(vector<int> distr, vector<preAllocLeaf> &potentialLeaves, Node *cur);
bool lessTs(Node *n1, Node *n2);

#endif
```

```
//Binding
#ifndef BIND_H
#define BIND_H

#include <token.h>
#include <map>
#include <utility>
#include <fstream.h>

struct property {
int delay;
int energy;
int area;
};

typedef map<int, property, std::less<int> > implib;
// implementation lib for an operation
// first refers to the bind, second to the delay and energy.
typedef map<int, implib, std::less<int> >  toplib;
// collection of implementation libs for all operations
// first int refers to the operator_kind

// functions associated with binding
toplib buildlib(ifstream &input);
int getDelay(int opkind, int bind,  toplib &mylib);
int getEnergy(int opkind, int bind,  toplib &mylib);
int getArea(int opkind, int bind, toplib &mylib);
int getDefaultBind(int opkind,  toplib &mylib);
int getDefaultDelay(int opkind,  toplib &mylib);
int getDefaultEng(int opkind,  toplib &mylib);
int getDefaultArea(int opkind, toplib &mylib);
#endif
```

```
//Chaining
#ifndef CHAIN_H
```

```
#define CHAIN_H

#include "component.h"
#include "asapsch.h"

void dual0p2chain0p(Subgraph *subg, opSource &available);
void single0p2chain0p(Subgraph *subg, opSource &available);

#endif
```

```
//The classes needed to describe clock tree such as clock leaf and clkEdges
#ifndef COMPONENT_H
#define COMPONENT_H

#define engCycle 60
#include "extInfo.h"
#include <extcdfg.h>
#include <string>
#include <vector>
#include "bind.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <map>
#include <math.h>
#include <list>
#include <iomanip.h>
#define NoneEx -1
class preAllocLeaf;
class leaf;
class clkEdge;
class clkLevel;
class pairClkEdge;
class pairClkEdgeList;
class clkTree;

typedef map<int, vector<Node *>, std::less<int> >  potentialNodes;
// int refers to the bind
int getLevel(int size);
int interTreeEnergy( clkTree &tree1, clkTree &tree2);
int getWeight( int base, int size);
bool overlap(Node *, Node *, int, toplib &, int);
NodePtrList  getPredNodePtrList(Node *n);
int getTsHead(NodePtrList &preNodes);
NodePtrList  getSuccNodePtrList(Node *n);
int getTsEnd(Node *n, NodePtrList &succNodes);
clkLevel getBottomLev(vector<leaf> &baseLeaves, toplib &mylib, int clock);
vector<leaf> rearrange( vector<leaf> &base, toplib &mylib, int clock);

class preAllocLeaf: public potentialNodes
{
 public:
  preAllocLeaf(int=0, int=0, int=0);
  preAllocLeaf(int, int, int, potentialNodes &list);
  int getTend( int clock, toplib &mylib);
  vector<Node *> getAllNodes();
  int setOpkind( int o) { return opkind=o; }
  int getOpkind() { return opkind; }
  int setSeq(int s) { return seq=s; }
  int getSeq() { return seq; }
  int setBind(int b) { return bind=b; }
  int getBind() {  return bind; }
 protected:
  int opkind;
  int seq;
  int bind;
```

```
};

class leaf : public preAllocLeaf
{
 public:
  leaf(): preAllocLeaf(), assign() {}
  leaf(preAllocLeaf &base):preAllocLeaf(base),assign() {}
  clkEdge bottomClkEdge(int clock);
  vector<Node *> setAssign(vector<Node *> &right)
    { return assign=right; }
  vector<Node *> &getAssign() { return assign; }
  int leafEnergy(toplib &mylib);
  void appAssign(Node *nodePtr)
    { assign.push_back(nodePtr); }
  void makeConcur(leaf &right, int clock);
 private:
  vector<Node *> assign;
};

class clkEdge: public vector<bool>
{
  friend ostream &operator<<(ostream &output, const clkEdge &clkEdge1);
 public:
  clkEdge(int length=0);
  clkEdge(vector<bool> &actPattern);
  int getParentSeq() { return parentSeq; }
  int setParentSeq(int s) { return parentSeq=s; }
  int getLeftSeq() { return leftSeq; }
  int setLeftSeq(int s) { return leftSeq=s; }
  int getRightSeq() { return leftSeq; }
  int setRightSeq(int s ) { return rightSeq=s; }
  int setSeq(int s) { return seq=s; }
  int getSeq() { return seq; }
  clkEdge upperClkEdge( );
  clkEdge upperClkEdge(clkEdge &sibling);
  int activeCycle();
  int edgeEnergy(int weight);
  int logicDistance(const clkEdge &clkEdge1);
 protected:
  int seq;
  int leftSeq;
  int rightSeq;
  int parentSeq;
};

class clkLevel: public vector<clkEdge>
{
 public:
  clkLevel(): vector<clkEdge>() {}
  clkLevel(const clkLevel &right);
  clkLevel  parentClkLevel();
  int clkLevelEnergy(int weight);
  void outputFile(ofstream &output);
  int logDisLevel();
  clkEdge* findBySeq(int m);
};

class pairClkEdge
{
  friend ostream &operator<<(ostream &output, const pairClkEdge &clkEdge1);
 public:
  pairClkEdge(int d=0, int s1=0, int s2=0): logicDis(d), seq1(s1), seq2(s2) {}
  pairClkEdge(const pairClkEdge &right): logicDis(right.logicDis),
    seq1(right.seq1), seq2(right.seq2)  {}
  int setLogicDis(int d) { return logicDis=d; }
  int getLogicDis() { return logicDis; }
  int setSeq1(int s1) { return seq1=s1; }
  int getSeq1() { return seq1; }
  int setSeq2(int s2) { return seq2=s2; }
  int getSeq2() { return seq2; }
  bool operator<(const pairClkEdge &comp);
```

```
  pairClkEdge &operator=(const pairClkEdge &right);
  bool overlap(pairClkEdge &p1);
 private:
  int logicDis;
  int seq1;
  int seq2;
};

class pairClkEdgeList: public list<pairClkEdge>
{
 public:
  pairClkEdgeList():list<pairClkEdge>() {}
  void buildSortedList(clkLevel &lev);
  void getPairList();
  int unPairClkEdge(const clkLevel &lev);
  void appendUnpair(int unpair);
  clkLevel pairedClkLevel(clkLevel &lev);
  void outputFile(ofstream &output);
};

class clkTree: public vector<clkLevel>
{
 public:
  clkTree() {}
  void buildTree(clkLevel bottom);
  void buildTreeRandomly(clkLevel bottom);
  int  treeEnergy();
  void outputFile(ofstream &output);
};

#endif
```

```
//Energy on CDFG
#ifndef ENERGY_H
#define ENERGY_H

#include "component.h"
#include "asapsch.h"
#include <algorithm>

int getAveLeafEnergy(vector<leaf> &base, toplib &mylib);
int getAveTreeEnergy(vector<leaf> &base, toplib &mylib, int clock );
int AfterReschLeafEnergy(vector<leaf> &base, toplib &mylib, int clock );
int AfterReschTreeEnergy ( vector<leaf> &base, toplib &mylib, int clock );
int getArea(opSource &available, toplib &mylib, int clock);
//int getAveTotEnergy(vector<leaf> &base, toplib &mylib, int clock );
//int energyAfterResch( vector<leaf> &base, toplib &mylib, int clock );

#endif
```

```
//Extend original node
#define NO_EXT_EDGE
#define NO_EXT_SUBG
#define NO_PARSE_INIT
#include <cdfg.h>
// addadd1 refers to chaining of 2 additions with 1 output
// addadd2 refers to chaining of 2 additions with 2 outputs
// e.g. chain operator (a+b+c)  with output of  (a+b) as well
// as (a+b+c)
```

```
struct ExtNodeInfo
{
  int bind;            //binding of operation
  int ts;              //time of the start
  int tend;            //time of the end
  int energy;          //energy consumed by the operation
  NodePtrList vtualPreds;
  //the virtual preceding nodes due to limited resource when asap
  NodePtrList vtualSuccs;
  //the virtual succeding nodes due to limited resource when alap
  int asapTs;
  int asapTend;
  int alapTs;
  int alapTend;

};

//int name2chainOp(const char *name);

#include <extcdfg.h>

#define CHKMARK(n,m) ((n)->getMark() & (m))
#define SETMARK(n,m) (n)->setMark((n)->getMark() | (m))
#define CLRMARK(n,m) (n)->setMark((n)->getMark() & ~(m))

#endif




//Multiple clocks
#ifndef MULTICLK_H
#define MULTICLK_H

#include "asapsch.h"

void ASAPMultiClk(Subgraph *subg, int ts, srcByCycle &hardware, const opSource &available, toplib
&mylib, int clock1, int clock2, Subgraph *root);

#endif



//Determine the resource necessary to CDFG
#ifndef RESOURCE_H
#define RESOURCE_H

#include "component.h"
#include "asapsch.h"
//#include "critpath.h"

typedef map<int , int, std::less<int> > bindAndNum;
// first int refers to bind, second to number of modules
typedef map<int, bindAndNum, std::less<int> > srcLib;
// first int refers to op_kind

void  getPotentialNodes(Subgraph *subg, int opKind, int bind, preAllocLeaf &associate);
void getNodes(Subgraph *subg, int opKind, int bind, vector<Node *> &vectNodes);
int getQuantity(vector<Node*> &vectNodes);
srcLib getSrcLib(Subgraph *subg, toplib &mylib, opSource &hardwareSrs);
vector<preAllocLeaf> getModules(srcLib &src, Subgraph *subg);

#endif
```

```
//mainf.cc
#include "asapsch.h"
#include <stdlib.h>
#include "component.h"
#include "assign.h"
#include "resource.h"
#include "energy.h"
#include "chain.h"
#include "multiclk.h"
void propertyUtil( Subgraph *root, toplib &mylib, opSource &available, int clock);
void resch(const char *cdfgFile, const char *cstr, toplib &mylib, int clock);
void chain(const char *cdfgFile, const char *cstr, toplib &mylib, int clock);
void multiClk(const char *cdfgFile, const char *cstr, const char *cstr1, const char *cstr2,
              toplib &mylib, int clock1, int clock2);

int main(int argc, char **argv)
{
  if(argc<6) return 0;
  ifstream lib("bindlib.txt", ios::in);
  if( !lib )
  {
    cerr<<"File could not be opened\n";
    exit(1);
  }
  toplib mylib=buildlib(lib);
  srand(time(NULL));

  resch(argv[1], argv[2], mylib, 11);
  chain(argv[1], argv[3], mylib, 22);
   multiClk(argv[1], argv[2], argv[4], argv[5], mylib, 11, 22);
  return 1;
}

void propertyUtil( Subgraph *root, toplib &mylib, opSource &available, int clock)
{
  srcLib  src=getSrcLib(root, mylib, available);
  vector<preAllocLeaf> modules=getModules(src, root);
  vector<leaf> base=assignOpKind( modules, mylib, clock, available);
  int engPreoptLeaf=getAveLeafEnergy(base, mylib);
  int engPreoptTree=getAveTreeEnergy(base, mylib, clock );
  int engPostoptLeaf=AfterReschLeafEnergy(base, mylib, clock );
  int engPostoptTree=AfterReschTreeEnergy(base, mylib, clock );
  int area=getArea(available, mylib, clock );
  cout<<engPreoptLeaf<<'\t'<<engPreoptTree<<endl;
  cout<<engPostoptLeaf<<'\t'<<engPostoptTree<<endl;
  cout<<area<<endl;
}

void resch(const char *cdfgFile, const char *cstr, toplib &mylib, int clock)
{
  ifstream constraint(cstr,ios::in);
  if( !constraint )
  {
    cerr<<"File could not be opened\n";
    exit(1);
  }
  ExtCDFG graph;
  opSource available=hardwareSrs(constraint);
  srcByCycle hardware;
  hardware.insert(srcByCycle::value_type(0, available));
  graph.parse(cdfgFile);
  Subgraph *root=graph.getSubg();
  ASAP(root,0 ,hardware, available, mylib, clock, root);
  printAllNodes(graph.getSubg());
  propertyUtil( root, mylib, available, clock);
}

void chain(const char *cdfgFile, const char *cstr, toplib &mylib, int clock)
{
  ExtCDFG graph;
```

```
  graph.parse(cdfgFile);
  ifstream constraint(cstr,ios::in);
  if( !constraint )
    {
      cerr<<"File could not be opened\n";
      exit(1);
    }
  opSource available=hardwareSrs(constraint);
  dualOp2chainOp(graph.getSubg(), available);
  singleOp2chainOp(graph.getSubg(), available);
  Subgraph *root=graph.getSubg();
  srcByCycle hardware;
  hardware.insert(srcByCycle::value_type(0, available));
  ASAP(root,0 ,hardware, available, mylib, clock, root);
  printAllNodes(graph.getSubg());
  propertyUtil( root, mylib, available, clock);
}

void multiClk(const char *cdfgFile, const char *cstr, const char *cstr1, const char *cstr2,
              toplib &mylib, int clock1, int clock2)
{
  ifstream constraint(cstr,ios::in);
  if( !constraint )
  {
    cerr<<"File could not be opened\n";
    exit(1);
  }
  opSource available=hardwareSrs(constraint);

  ExtCDFG graph;
  graph.parse(cdfgFile);
  srcByCycle hardware;
  hardware.insert(srcByCycle::value_type(0, available));
  Subgraph *root=graph.getSubg();
  ASAPMultiClk(root, 0, hardware, available, mylib, clock1, clock2, root);
  printAllNodes(graph.getSubg());
  ifstream constraint1(cstr1,ios::in);
  if( !constraint1 )
  {
    cerr<<"File could not be opened\n";
    exit(1);
  }
  opSource available1=hardwareSrs(constraint1);
  ifstream constraint2(cstr2,ios::in);
  if( !constraint2 )
  {
    cerr<<"File could not be opened\n";
    exit(1);
  }
  opSource available2=hardwareSrs(constraint2);

  srcLib  src1=getSrcLib(graph.getSubg(), mylib, available1);
  vector<preAllocLeaf> modules1=getModules(src1, graph.getSubg());
  vector<leaf> base1=assignOpKind( modules1, mylib, clock1, available1);
  int engPostoptLeaf1=AfterReschLeafEnergy(base1, mylib, clock2 );
  int engPostoptTree1=AfterReschTreeEnergy(base1, mylib, clock2 );
  int area1=getArea(available1, mylib, clock2);
  cout<<engPostoptLeaf1<<'\t'<<engPostoptTree1<<'\t'<<area1<<endl;
  vector<leaf> after1=rearrange(base1, mylib, clock2);
  clkLevel bottomLev1=getBottomLev(after1, mylib, clock2);
  clkTree myTree1;
  myTree1.buildTree(bottomLev1);

  srcLib  src2=getSrcLib(graph.getSubg(), mylib, available2);
  vector<preAllocLeaf> modules2=getModules(src2, graph.getSubg());
  vector<leaf> base2=assignOpKind( modules2, mylib, clock1, available2);
  int engPostoptLeaf2=AfterReschLeafEnergy(base2, mylib, clock1 );
  int engPostoptTree2=AfterReschTreeEnergy(base2, mylib, clock1 );
  int area2=getArea(available2, mylib, clock1);
  cout<<engPostoptLeaf2<<'\t'<<engPostoptTree2<<'\t'<<area2<<endl;
```

```
  vector<leaf> after2=rearrange(base2, mylib, clock1);
  clkLevel bottomLev2=getBottomLev(after2, mylib, clock1);
  clkTree myTree2;
  myTree2.buildTree(bottomLev2);
  int interTreeEng=interTreeEnergy(myTree1, myTree2);
  cout<<interTreeEng<<endl;

  cout<<(engPostoptLeaf1+engPostoptLeaf2)<<'\t'
      <<(engPostoptTree1+engPostoptTree2+interTreeEng)<<'\t'
      <<(area1+area2)<<endl;
}
```

REFERENCES

[1] M. J. Riezenman, "The search for better batteries," in IEEE Spectrum, Volume: 32, Issue: 5, pp. 51-56, May 1995.

[2] D. Dobberpuhl et al, "A 200MHz, 64bit, dual issue CMOS microprocessor," Digest of Technical Paper, ISSC'92, pp. 106-107

[3] M. Pedram, "Power minimization in IC design: Principles and applications," ACM Trans. Design Automation Electronic Systems, vol. 1, pp. 3-56, Jan. 1996

[4] C. Small, "Shrinking devices put the squeeze on system packaging," Electronic Design News, vol.39, pp. 41-46, Feb. 1994

[5] P. Yang and J.H. Chen, "Design for reliability: The major challenge for VLSI," Proc. IEEE, vol. 81, pp. 730-744, May 1993

[6] B. Nadel, "The green machine," PC Magazine, vol. 12, May 1993

[7] H. Veendrick, " Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits," IEEE, J. Solid State Circuits 19, pp. 468-473, Aug. 1984

[8] N. Hedenstierna and K. Jeppson, " CMOS circuit speed and buffer optimization," IEEE tran., Computer-Aided Design, Integrated Circuit Sys. 6, pp. 270-281, Mar, 1987

[9] A. Bellaouar and M.I. Elmasry, Low-Power Digital VLSI design - Circuits and Systems, Kluwer Academic Publishers, Norwell, MA, 1995.

[10] M. Horowitz, T. Indermaur, and R. Gonzalez, " Low-power digital design," in Proc. Symp. Low Power Electronics, pp. 8-11, Oct. 1994

[11] A. R. Chandrakasan and R.W. Brodersen, Low Power Digital CMOS Design. Kluwer Academic Publishers, Norwell, MA, 1995

[12] L. Goodby, A. Orailoglu, and P. M. Chau, " Microarchitectural synthesis of performance-constrained, low power VLSI design," in Proc. Int. Conf. Computer Design, pp. 323-326, Oct. 1994

[13] R. S. Martin and J. P. Knight, "Power Profiler: Optimizing ASICs power consumption at the behavioral level," in Proc. Design Automation Conf. , pp. 42-47, June 1995

[14] A. Raghunathan and N. K. Jha, " An iterative improvement algorithm for low power data path synthesis," in Proc. Int. Conf. Computer-Aided Design, pp. 597-602, Nov. 1995

[15] S. Raje and M. Sarrafzadeh, "Variable voltage scheduling," in Proc. Int. Symp. Low Power Design, pp. 9-14, Apr. 1995

[16] J. M. Chang and M. Pedram, "Energy minimization using multiple supply voltages," in Proc. Int. Symp. Low Power Electronics & Design, pp. 157-162, Aug. 1996

[17] S. Devadas and S. Malik, " A Survey of Optimization Techniques targeting low power VLSI circuits," in Proc. Design Automation Conf., pp. 242-247, June 1995

[18] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low power: an overview," In Proc. Symp. On Low Power Electronics, pp. 38-39, Oct. 1994

[19] D. Yuen, "Intel's SL Architecture - Designing Portable Applications," McGraw-Hill, New York, NY, 1993

[20] http://poppy.snu.ac.kr/vdt-ivsim/vhdlsim.html

[21] http://poppy.snu.ac.kr/~shleee/class/icda2001/CDFGTool.pdf

[22] R. A. Walker and S. Chaudhuri, "High-level synthesis: Introduction to the scheduling problem," IEEE trans. Design & Test of Computers, Vol. 12, issue 2, pp. 60-69, 1995

[23] B. M. Pangrle and D. D. Gajski, "Design tools for intelligent silicon compilation," IEEE trans. On CAD, pages 1098-1112, Nov. 1987

[24] P. G. Paulin and J. P. Knight, " Algorithms for high-level synthesis," IEEE Design and Test, pp. 18-31, Dec. 1989

[25] K. Usami and M. Horowitz, " Clustered voltage scaling technique for low power design, " in proc. Int. Symp. Low Power Design, pp. 3-8, Apr. 1995

[26] A. Raghunathan, N. K. Jha, and S. Dey, "High-level power analysis and optimization," Kluwer Academic Publishers, 1998

[27] A. Raghunathan and N. K. Jha, "Behavioral synthesis for low power," in Proc. Int. Conf. Computer Design, pp. 318-322, Oct. 1994

[28] A. Raghunathan and N. K. Jha, "An ILP formulation for low power based on minimizing switched capacitance during datapath allocation," in Proc. Int. Symp. Circuit & Systems, pp. 1069-1073, May 1995

[29] E. Musoll and J. Cortadella, "High-level synthesis techniques for reducing the activity of functional units," in Proc. Int. Symp. Low Power Design, pp. 99-104, Apr. 1995

[30] R. Meha and J. Rabaey, "Exploiting regularity for low-power design," in Proc. Int. Conf. Computer-Aided Design, pp. 166-172, Nov. 1996

[31] M. Edahiro, "Delay minimization for zero-skew routing," Research report CS-TR-415-93, Princeton Univ. Mar. 1993

[32] http://eda.ece.wisc.edu/ECE902/int_delay1. PDF

[33] W.C. Elmore, " The transit response of damped linear networks with particular regard to wideband amplifiers," Journal of Applied Physics, Vol. 10, pp. 55-63, 1948)

[34] J. Rubinstein, P. Penfield. Jr., and M. A. Horowitz, " Signal delay in RC tree networks," IEEE trans. Computer-Aided Design, vol. 2, No. 3, pp. 202-211, July, 1983

[35] E. G. Friedman, "Clock distribution design in VLSI circuits – an overview," in Proc. Int. Symp. On Circuits and System, pp. 1475-1478, May, 1993

[36] J. P. Fishburn, "Clock skew optimization," IEEE trans. on Computers, Vol. C-39, No. 7, pp. 945-951, July 1990

[37] D. F. Wann and M. A. Franklin, " Asynchronous and clocked control structures for VLSI based interconnection networks," IEEE trans. on Computers, vol. C-32, No. 3, pp 284-293, Mar. 1983.

[38] M. A. B. Jackson, A. Srinivasan and E. S. Kuh, "Clock routing for high-performance Ics," in Proc. Design Automation conf. pp. 573-679, 1990

[39] R. S. Tsay, "An exact zero-skew clock routing algorithm," IEEE trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. 12, No. 2, Feb. 1993

[40] K. D. Boese and A. B. Kahng, "Zero-skew clock routing trees with minimum wirelength," in Proc. IEEE 5th Intl. ASIC Conf. pp. 111-115, Sept, 1992

[41] C. Papachristou, M. Spining, and M. Nourani, "An effective power management scheme for RTL design based on multiple clocks," in Proc. Design Automation Conf. pp. 337-342, June 1996

[42] M. Igarashi, K. Usami, K. Nogami and F. Minami et al, "A low-power design method using multiple supply voltages," in Proc. Of the ACM int. symp. on Low Power Electronics and Design, pp. 36-41, 1997

[43] J. Oh and M. Pedram, "Gated clock routing for low-power microprocessor design," in IEEE trans. on Computer-Aided Design of Integrated and Systems, Vol. 20, No. 6, pp. 715-722, June 2001

[44] A. H. Farrahi, C. H. Chen, M. Sarrafzadeh, and G. Tellez, "Activity-driven clock design," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 20, no. 6, pp. 706-714, June 2001.

[45] J. Panjun and S. S. Sapatnekar, "Clock distribution using multiple voltages", in Proceedings of the *ACM International Symposium on Low Power Electronics and Design*, pp.145-150, 1999.

[46] N. C. Chou and C. K. Cheng, "Wire length and delay minimization in general clock net routing", in Proceedings of *International Conference on Computer-Aided Design*, pp. 118-122, 1993.

[47] D. Garrett, M. Stan, A. Dean, "Challenges in clock gating for a low power ASIC methodology", in Proceedings of the ACM International Symposium on Low Power Electronics and Design, pp. 176 –181, 1999.

[48] M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak, and J. Rabaey, "Instruction set mapping for performance optimization," in Proceedings of International Conference on Computer-Aided Design, pp.518-521, 1993.

VITA AUCTORIS

NAME:	Changjun Kang	
DATE OF BIRTH:	14th December, 1968	
PLACE OF BIRTH:	Nanchang, Jiangxi, China	
EDUCATION:	1980-1982	Jiangzhong School
	1982-1986	Chongren #1 High School
	1986-1991	Jilin University (B. S.)
	1991-1994	Shanghai Institute of Metallurgy, Chinese Academy of Science (M.S.)
	2000-2002	University of Windsor (M.A. Sc.)
WORK HISTORY	1994-1996	Integrated Circuit Design Engineer, Sino-Wealth Electronic Ltd.
	1996-1997	Field Application Engineer, National Semiconductor
	1997-1999	Field Application Engineer, Temic Semiconductor